VMS Software

VSI C
User Manual

Operating System and Version: VSI OpenVMS Alpha Version 8.4-2L1 or higher
VSI OpenVMS IA-64 Version 8.4-1H1 or higher

Software Version: VSI C Version 7.4-1 for OpenVMS Alpha
VSI C Version 7.4-1 for OpenVMS IA-64

VMS Software, Inc. (VSI)
Boston, Massachusetts, USA

VSI C User Manual

Copyright © 2026 VMS Software, Inc. (VSI), Boston, Massachusetts, USA

Legal Notice

Confidential computer software. Valid license from VSI required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

The information contained herein is subject to change without notice. The only warranties for VSI products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. VSI shall not be liable for technical or editorial errors or omissions contained herein.

HPE, HPE Integrity, HPE Alpha, and HPE Proliant are trademarks or registered trademarks of Hewlett Packard Enterprise.

Intel, Itanium and IA64 are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java, the coffee cup logo, and all Java based marks are trademarks or registered trademarks of Oracle Corporation in the United States or other countries.

Kerberos is a trademark of the Massachusetts Institute of Technology.

Microsoft, Windows, Windows-NT and Microsoft XP are U.S. registered trademarks of Microsoft Corporation. Microsoft Vista is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries.

Motif is a registered trademark of The Open Group.

UNIX is a registered trademark of The Open Group.

Table of Contents

Preface xiii
1. About VSI xiii
2. Intended Audience xiii
3. Document Structure xiii
4. Related Documents xiv
5. OpenVMS Documentation xiv
6. VSI Encourages Your Comments xiv
7. Platform Labels xv
8. Typographical Conventions xv
9. New and Changed Features xv

Chapter 1. Developing VSI C Programs 1
1.1. DCL Commands for Program Development 1
1.2. Creating a VSI C Program 3
1.2.1. Using TPU 3
1.2.2. The EVE Interface to TPU 3
1.3. Compiling a VSI C Program 3
1.3.1. The CC Command 4
1.3.1.1. Including Header Files 5
1.3.1.2. Listing Header Files 6
1.3.2. Compilation Modes 7
1.3.3. Microsoft Compatibility Compilation Mode 9
1.3.3.1. Unnamed Nested struct or union Members 9
1.3.3.2. Block Scope Declaration of static Functions 9
1.3.3.3. Treat &* as Having No Effect 9
1.3.3.4. char is Not Treated as a Unique Type 9
1.3.3.5. Double Semicolons in Declarations 10
1.3.3.6. Declaration without a Type 10
1.3.3.7. Enumerators in an Enumeration Declaration 10
1.3.3.8. Useless Typedefs 10
1.3.3.9. Unrecognized Pragmas Accepted 10
1.3.4. CC Command Qualifiers 10
1.3.5. Compiler Diagnostic Messages 61
1.4. Linking a VSI C Program 62
1.4.1. The LINK Command 63
1.4.2. LINK Command Qualifiers 63
1.4.3. Linker Input Files 65
1.4.4. Linker Output Files 65
1.4.5. Linking Against Object Module Libraries and Shareable Images 65
1.4.6. Object Module Libraries 66
1.4.7. Linker Error Messages 66
1.5. Running a VSI C Program 67
1.6. Passing Arguments to the main Function 69
1.7. 64-bit Addressing Support 71
1.7.1. Qualifiers and Pragmas 71
1.7.1.1. The /POINTER_SIZE Qualifier 71
1.7.1.2. The __INITIAL_POINTER_SIZE Macro 72
1.7.1.3. The /CHECK=POINTER_SIZE Qualifier 72
1.7.1.4. Pragmas 73
1.7.2. Determining Pointer Size 73

1.7.2.1. Special Cases 74
1.7.2.2. Mixing Pointer Sizes 75
1.7.3. Header File Considerations 75
1.7.4. Prologue/Epilogue Files 76
1.7.4.1. Rationale 76
1.7.4.2. Using Prologue/Epilogue Files 77
1.7.5. Avoiding Problems 78
1.7.6. Examples 78
Chapter 2. Using OpenVMS Record Management Services 83
2.1. RMS File Organization 83
2.1.1. Sequential File Organization 84
2.1.2. Relative File Organization 84
2.1.3. Indexed File Organization 85
2.2. Record Access Modes 85
2.3. RMS Record Formats 85
2.4. RMS Functions 86
2.5. Writing VSI C Programs Using RMS 87
2.5.1. Initializing File Access Blocks 89
2.5.2. Initializing Record Access Blocks 89
2.5.3. Initializing Extended Attribute Blocks 90
2.5.4. Initializing Name Blocks 91
2.6. RMS Example Program 91
Chapter 3. Using VSI C in the Common Language Environment 107
3.1. Basic Calling Standard Conventions 108
3.1.1. Register and Stack Usage 108
3.1.2. Return of the Function Value 110
3.1.3. The Argument List 110
3.2. Specifying Parameter-Passing Mechanisms 111
3.2.1. Passing Arguments by Immediate Value 111
3.2.2. Passing Arguments by Reference 114
3.2.3. Passing Arguments by Descriptor 116
3.2.4. VSI C Default Parameter-Passing Mechanisms 120
3.3. Interlanguage Calling 121
3.3.1. Calling FORTRAN 121
3.3.2. Calling VAX MACRO 126
3.3.3. Calling VSI BASIC 129
3.3.4. Calling VSI Pascal 131
3.4. Sharing Global Data 136
3.4.1. Sharing Program Sections with FORTRAN Common Blocks 136
3.4.2. Sharing Program Sections with PL/I Externals 138
3.4.3. Sharing Program Sections with MACRO Programs 140
3.5. OpenVMS Run-Time Library Routines 141
3.6. OpenVMS System Services Routines 142
3.7. Calling Routines 142
3.7.1. Determining the Type of Call 143
3.7.2. Declaring an External Routine and Its Arguments 143
3.7.3. Calling the External Routine 143
3.7.4. System Routine Arguments 143
3.7.5. Symbol Definitions 146
3.7.6. Condition Values 147
3.7.7. Checking System Service Return Values 147

3.8. Variable-Length Argument Lists in System Services 148
3.9. Return Status Values 150
3.9.1. Format of Return Status Values 150
3.9.2. Manipulating Return Status Values 152
3.9.3. Testing for Success or Failure 153
3.9.4. Testing for Specific Return Status Values 154
3.10. Examples of Calling System Routines 155
Chapter 4. Data Storage and Representation 159
4.1. Storage Allocation 159
4.2. Standard-Conforming Method of Controlling External Objects 160
4.3. Global Storage Classes 161
4.3.1. The globaldef and globalref Specifiers 161
4.3.2. Comparing the Global and the External Storage Classes 163
4.3.3. The globalvalue Specifier 165
4.4. Storage-Class Modifiers 165
4.4.1. The noshare Modifier 166
4.4.2. The readonly Modifier 166
4.4.3. The _align Modifier 167
4.5. Floating-Point Numbers (float, double, long double) 167
4.6. Pointer Conversions 169
4.7. Structure Alignment 169
4.7.1. Bit-Field Alignment 170
4.7.2. Bit-Field Initialization 171
4.7.3. Variant Structures and Unions 171
4.8. Program Sections 173
4.8.1. Attributes of Program Sections 173
4.8.2. Program Sections Created by VSI C 174
Chapter 5. Preprocessor Directives 181
5.1. CDD/Repository Extraction (#dictionary) 181
5.2. File Inclusion (#include) 181
5.2.1. Inclusion Using Angle Brackets 182
5.2.2. Inclusion Using Quotation Marks 183
5.2.3. Inclusion of Text Modules 185
5.2.4. Macro Substitution in #include Directives 185
5.3. Changing the Default Object Module Name and Identification (#module) 186
5.4. Implementation-Specific Preprocessor Directive (#pragma) 186
5.4.1. #pragma assert Directive 187
5.4.1.1. #pragma assert func_attrs 187
5.4.1.2. #pragma assert global_status_variable 188
5.4.1.3. Usage Notes 188
5.4.1.4. #pragma assert non_zero 189
5.4.2. #pragma builtins Directive 189
5.4.3. #pragma dictionary Directive 190
5.4.4. #pragma environment Directive 192
5.4.5. #pragma extern_model Directive 193
5.4.5.1. Syntax 195
5.4.5.2. #pragma extern_model common_block 197
5.4.5.3. #pragma extern_model relaxed_refdef 197
5.4.5.4. #pragma extern_model strict_refdef 198
5.4.5.5. #pragma extern_model globalvalue 199
5.4.5.6. #pragma extern_model save 199

5.4.5.7. #pragma extern_model restore 199
5.4.5.8. Effects on the VSI C Run-Time Library and User Programs 199
5.4.5.9. Example 201
5.4.6. #pragma extern_prefix Directive 202
5.4.7. #pragma function Directive 203
5.4.8. #pragma [no]include_directory Directive 204
5.4.9. #pragma [no]inline Directive 204
5.4.10. #pragma intrinsic Directive 206
5.4.11. #pragma linkage Directive (Alpha only) 207
5.4.12. #pragma linkage Directive (I64 only) 210
5.4.12.1. #pragma linkage Format 210
5.4.12.2. #pragma linkage_ia64 Format 212
5.4.13. #pragma [no]member_alignment Directive 212
5.4.14. #pragma message Directive 214
5.4.14.1. #pragma message *option1* 214
5.4.14.2. #pragma message *option2* 217
5.4.14.3. #pragma message (*quoted-string*) 217
5.4.15. #pragma module Directive 217
5.4.16. #pragma names Directive 218
5.4.17. #pragma optimize Directive 219
5.4.18. #pragma pack Directive 220
5.4.19. #pragma pointer_size Directive 222
5.4.20. #pragma required_pointer_size Directive 222
5.4.21. #pragma [no]standard Directive 223
5.4.22. #pragma unroll Directive 224
5.4.23. #pragma use_linkage Directive 224

Chapter 6. Predefined Macros and Built-In Functions 227

6.1. Predefined Macros 227
6.1.1. CC$gfloat (G_Floating Identification Macro) 227
6.1.2. System Identification Macros 227
6.1.2.1. The __DECC_VER Macro 229
6.1.2.2. The __VMS_VER Macro 230
6.1.3. Standards Conformance Macros 231
6.1.4. Floating-Point Macros 232
6.1.5. Compiler-Mode Macros 233
6.1.6. Pointer-Size Macro 233
6.1.7. The __HIDE_FORBIDDEN_NAMES Macro 233
6.2. Built-In Functions 234
6.2.1. Built-In Functions for OpenVMS Alpha Systems (Alpha only) 235
6.2.1.1. Translation Macros for VAX C Built-in Functions 235
6.2.1.2. In-line Assembly Code – ASMs 235
6.2.1.3. Absolute Value (__ABS) 238
6.2.1.4. Acquire and Release Longword Semaphore (__ACQUIRE_SEM_LONG, __RELEASE_SEM_LONG) 238
6.2.1.5. Add Aligned Word Interlocked (__ADAWI) 239
6.2.1.6. Add Atomic Longword (__ADD_ATOMIC_LONG) 240
6.2.1.7. Add Atomic Quadword (__ADD_ATOMIC_QUAD) 240
6.2.1.8. Allocate Bytes from Stack (__ALLOCA) 241
6.2.1.9. AND Atomic Longword (__AND_ATOMIC_LONG) 241
6.2.1.10. AND Atomic Quadword (__AND_ATOMIC_QUAD) 242
6.2.1.11. Atomic Add Longword (__ATOMIC_ADD_LONG) 242
6.2.1.12. Atomic Add Quadword (__ATOMIC_ADD_QUAD) 243

6.2.1.13. Atomic AND Longword (__ATOMIC_AND_LONG) 244
6.2.1.14. Atomic AND Quadword (__ATOMIC_AND_QUAD) 244
6.2.1.15. Atomic OR Longword (__ATOMIC_OR_LONG) 245
6.2.1.16. Atomic OR Quadword (__ATOMIC_OR_QUAD) 245
6.2.1.17. Atomic Increment Longword (__ATOMIC_INCREMENT_LONG) 246
6.2.1.18. Atomic Increment Quadword (__ATOMIC_INCREMENT_QUAD) 247
6.2.1.19. Atomic Decrement Longword (__ATOMIC_DECREMENT_LONG) 247
6.2.1.20. Atomic Decrement Quadword (__ATOMIC_DECREMENT_QUAD) 248
6.2.1.21. Atomic Exchange Longword (__ATOMIC_EXCH_LONG) 248
6.2.1.22. Atomic Exchange Quadword (__ATOMIC_EXCH_QUAD) 249
6.2.1.23. Compare Store Longword (__CMP_STORE_LONG) 249
6.2.1.24. Compare Store Quadword (__CMP_STORE_QUAD) 250
6.2.1.25. Convert G_Floating to F_Floating Chopped (__CVTGF_C) 250
6.2.1.26. Convert G_Floating to Quadword (__CVTGQ) 250
6.2.1.27. Convert IEEE T_Floating to IEEE S_Floating Chopped (__CVTTS_C) 250
6.2.1.28. Convert IEEE T_Floating to Quadword (__CVTTQ) 251
6.2.1.29. Convert X_Floating to Quadword (__CVTXQ) 251
6.2.1.30. Convert X_Floating to IEEE T_Floating Chopped (__CVTXT_C) 251
6.2.1.31. Copy Sign Built-in Functions 251
6.2.1.32. Cosine (__COS) 252
6.2.1.33. Double-Precision, Floating-Point Arithmetic Built-in Functions 252
6.2.1.34. Floating-Point Absolute Value (__FABS) 253
6.2.1.35. _leadz 253
6.2.1.36. Long Double-Precision, Floating-Point Arithmetic Built-in Functions 253
6.2.1.37. Longword Absolute Value (__LABS) 253
6.2.1.38. Lock and Unlock Longword (__LOCK_LONG, __UNLOCK_LONG) 254
6.2.1.39. Memory Barrier (__MB) 254
6.2.1.40. Memory Copy and Set Functions (__MEMCPY, __MEMMOVE, __MEMSET) 254
6.2.1.41. OR Atomic Longword (__OR_ATOMIC_LONG) 255
6.2.1.42. OR Atomic Quadword (__OR_ATOMIC_QUAD) 255
6.2.1.43. Privileged Architecture Library Code Instructions 256
6.2.1.44. __PAL_BPT 256
6.2.1.45. __PAL_BUGCHK 256
6.2.1.46. __PAL_CFLUSH 256
6.2.1.47. __PAL_CHME 257
6.2.1.48. __PAL_CHMK 257
6.2.1.49. __PAL_CHMS 257
6.2.1.50. __PAL_CHMU 257
6.2.1.51. __PAL_DRAINA 257
6.2.1.52. __PAL_GENTRAP 257
6.2.1.53. __PAL_HALT 258
6.2.1.54. __PAL_INSQHIL 258
6.2.1.55. __PAL_INSQHILR 258
6.2.1.56. __PAL_INSQHIQ 259
6.2.1.57. __PAL_INSQHIQR 259
6.2.1.58. __PAL_INSQTIL 260
6.2.1.59. __PAL_INSQTILR 260
6.2.1.60. __PAL_INSQTIQ 261
6.2.1.61. __PAL_INSQTIQR 261

6.2.1.62. __PAL_INSQUEL 262
6.2.1.63. __PAL_INSQUEL_D 262
6.2.1.64. __PAL_INSQUEQ 262
6.2.1.65. __PAL_INSQUEQ_D 263
6.2.1.66. __PAL_LDQP 263
6.2.1.67. __PAL_STQP 263
6.2.1.68. __PAL_MFPR_XXXX 264
6.2.1.69. __PAL_MTPR_XXXX 264
6.2.1.70. __PAL_PROBER 265
6.2.1.71. __PAL_PROBEW 265
6.2.1.72. __PAL_RD_PS 266
6.2.1.73. __PAL_REMQHIL 266
6.2.1.74. __PAL_REMQHILR 266
6.2.1.75. __PAL_REMQHIQ 267
6.2.1.76. __PAL_REMQHIQR 267
6.2.1.77. __PAL_REMQTIL 268
6.2.1.78. __PAL_REMQTILR 268
6.2.1.79. __PAL_REMQTIQ 269
6.2.1.80. __PAL_REMQTIQR 269
6.2.1.81. __PAL_REMQUEL 270
6.2.1.82. __PAL_REMQUEL_D 270
6.2.1.83. __PAL_REMQUEQ 271
6.2.1.84. __PAL_REMQUEQ_D 271
6.2.1.85. __PAL_SWPCTX 271
6.2.1.86. __PAL_SWASTEN 272
6.2.1.87. __PAL_WR_PS_SW 272
6.2.1.88. _popcnt 272
6.2.1.89. _poppar 272
6.2.1.90. Read Process Cycle Counter (__RPCC) 272
6.2.1.91. Sine (__SIN) 273
6.2.1.92. Single-Precision, Floating-Point Arithmetic Built-in Functions 273
6.2.1.93. Test for Bit Clear then Clear Bit Interlocked (__INTERLOCKED_TESTBITCC_QUAD) 273
6.2.1.94. Test for Bit Clear then Clear Bit Interlocked (__TESTBITCCI) 274
6.2.1.95. Test for Bit Set Then Set Bit Interlocked (__INTERLOCKED_TESTBITSS_QUAD) 274
6.2.1.96. Test for Bit Set then Set Bit Interlocked (__TESTBITSSI) 275
6.2.1.97. _trailz 275
6.2.1.98. Trap Barrier Instruction (__TRAPB) 276
6.2.1.99. Unsigned Quadword Multiply High (__UMULH) 276
6.2.2. Built-In Functions for IA-64 systems (I64 only) 276
6.2.2.1. Builtin Differences on IA-64 Systems 276
6.2.2.2. Built-in Functions Specific to IA-64 systems 277
6.2.2.3. Get Hardware Register Value (__getReg) 277
6.2.2.4. Set Hardware Register Value (__setReg) 279
6.2.2.5. Get Index Register Value (__getIndReg) 279
6.2.2.6. Set Index Register Value (__setIndReg) 280
6.2.2.7. Generate Break Instruction (__break) 280
6.2.2.8. Serialize Data (__dsrlz) 281
6.2.2.9. Flush Cache Instruction (__fc) 281
6.2.2.10. Flush Write Buffers (__fwb) 281
6.2.2.11. Invalidate ALAT (__invalat) 281

6.2.2.12. Invalidate ALAT (__invala) 281
6.2.2.13. Execute Serialize (__isrlz) 281
6.2.2.14. Insert Data Address Translation Cache (__itcd) 281
6.2.2.15. Insert Instruction Address Translation Cache (__itci) 282
6.2.2.16. Insert Data Translation Register (__itrd) 282
6.2.2.17. Insert Instruction Translation Register (__itri) 282
6.2.2.18. Purge Translation Cache Entry (__ptce) 283
6.2.2.19. Purge Global Translation Cache (__ptcg) 283
6.2.2.20. Purge Local Translation Cache (__ptcl) 283
6.2.2.21. Purge Global Translation Cache and ALAT (__ptcga) 283
6.2.2.22. Purge Data Translation Register (__ptrd) 284
6.2.2.23. Purge Instruction Translation Register (__ptri) 284
6.2.2.24. Reset System Mask (__rsm) 284
6.2.2.25. Reset User Mask (__rum) 285
6.2.2.26. Set System Mask (__ssm) 285
6.2.2.27. Set User Mask (__sum) 285
6.2.2.28. Enable Memory Synchronization (__synci) 285
6.2.2.29. Translation Hashed Entry Address (__thash) 285
6.2.2.30. Translation Hashed Entry Tag (__ttag) 286
6.2.2.31. Atomic Compare and Exchange
(_InterlockedCompareExchange_acq) 286
6.2.2.32. Atomic Compare and Exchange
(_InterlockedCompareExchange64_acq) 287
6.2.2.33. Atomic Compare and Exchange (_InterlockedCompareExchange_rel) 287
6.2.2.34. Atomic Compare and Exchange
(_InterlockedCompareExchange64_rel) 287
6.2.2.35. Conditional Atomic Compare and Exchange Longword
(__CMP_SWAP_LONG) 287
6.2.2.36. Conditional Atomic Compare and Exchange Quadword
(__CMP_SWAP_QUAD) 288
6.2.2.37. Conditional Atomic Compare and Exchange Longword with Acquire
Semantics (__CMP_SWAP_LONG_ACQ) 288
6.2.2.38. Conditional Atomic Compare and Exchange Quadword with Acquire
Semantics (__CMP_SWAP_QUAD_ACQ) 289
6.2.2.39. Conditional Atomic Compare and Exchange Longword with Release
Semantics (__CMP_SWAP_LONG_REL) 289
6.2.2.40. Conditional Atomic Compare and Exchange Quadword with Release
Semantics (__CMP_SWAP_QUAD_REL) 290
6.2.2.41. Return Address (__RETURN_ADDRESS) 290
6.2.2.42. Implement Alpha __PAL_GENTRAP and __PAL_BUGCHK Builtins
(__break2) 291
6.2.2.43. Flush Register Stack (__flushrs) 291
6.2.2.44. Load Register Stack (__loadrs) 291
6.2.2.45. Probe Read-Access Permission (__prober) 291
6.2.2.46. Probe Write-Access Permission (__probew) 292
6.2.2.47. Translation Access Key (__tak) 292
6.2.2.48. Translate to Physical Address (__tpa) 292
6.2.3. Built-In Functions for OpenVMS VAX Systems (VAX only) 293
6.2.3.1. Allocate Bytes from Stack (__ALLOCA) 293
6.2.3.2. Add Aligned Word Interlocked (_ADAWI) 293
6.2.3.3. Branch on Bit Clear-Clear Interlocked (_BBCCI) 293
6.2.3.4. Branch on Bit Set-Set Interlocked (_BBSSI) 294

6.2.3.5. Find First Clear Bit (_FFC) 294
6.2.3.6. Find First Set Bit (_FFS) 295
6.2.3.7. Halt (_HALT) 296
6.2.3.8. Insert Entry into Queue at Head Interlocked (_INSQHI) 296
6.2.3.9. Insert Entry into Queue at Tail Interlocked (_INSQTI) 296
6.2.3.10. Insert Entry in Queue (_INSQUE) 297
6.2.3.11. Locate Character (_LOCC) 297
6.2.3.12. Move from Processor Register (_MFPR) 298
6.2.3.13. Move Character 3 Operand (_MOVC3) 298
6.2.3.14. Move Character 5 Operand (_MOVC5) 299
6.2.3.15. Move from Processor Status Longword (_MOVPSL) 300
6.2.3.16. Move to Processor Register (_MTPR) 300
6.2.3.17. Probe Read Accessibility (_PROBER) 300
6.2.3.18. Probe Write Accessibility (_PROBEW) 301
6.2.3.19. Read General-Purpose Register (_READ_GPR) 301
6.2.3.20. Remove Entry from Queue at Head Interlocked (_REMQHI) 301
6.2.3.21. Remove Entry from Queue at Tail Interlocked (_REMQTI) 302
6.2.3.22. Remove Entry from Queue (_REMQUE) 303
6.2.3.23. Scan Characters (_SCANC) 303
6.2.3.24. Skip Character (_SKPC) 304
6.2.3.25. Span Characters (_SPANC) 304

Appendix A. Migrating from VAX C 307

A.1. Features Affecting the Compiler 307
A.1.1. VSI C Qualifiers 308
A.1.2. Comment Processing 309
A.1.3. String Literal Concatenation 309
A.1.4. Recursive main() Function 310
A.1.5. Trigraph Sequences 310
A.1.6. Alert Escape Sequence 310
A.1.7. Hexadecimal Escape Sequence 310
A.1.8. Invalid Escape Sequences 311
A.1.9. $ in Macro Names 311
A.1.10. Null Arguments to Macros 311
A.1.11. Standard C Name Space Conformance 311
A.1.11.1. Nonstandard Keywords 311
A.1.11.2. Nonstandard Predefined Macros 312
A.1.11.3. Nonstandard Identifiers in Standard-Specified Header Files 312
A.1.12. VSI C Predefined Macros 313
A.1.13. VSI C Types 313
A.1.13.1. signed Reserved Word 313
A.1.13.2. Removal of the long float Type 313
A.1.13.3. Addition of the long double Type 313
A.1.13.4. Addition of Processor-Specific Integer Data Types 314
A.1.14. Type Compatibility 315
A.1.15. Composite Types 315
A.1.16. Enumerations Have Type int 316
A.1.17. long double Constants 316
A.1.18. Implicit Unsigned Integer Constants 316
A.1.18.1. OpenVMS VAX Systems 316
A.1.18.2. OpenVMS Alpha Systems 316
A.1.19. Multibyte and Wide Character Support 317
A.1.19.1. The Wide Character Type 317

A.1.19.2. Multibyte Characters in Comments, Character Constants, and String Literals 317
A.1.19.3. Wide Character Constants 317
A.1.19.4. Wide String Literals 317
A.1.20. Usual Arithmetic Conversions 318
A.1.21. Indexing as a Commutative Operator 318
A.1.22. Cast Operators 318
A.1.23. Function Calls 318
A.1.23.1. Assignment Compatibility Argument Checking 318
A.1.23.2. Passing Narrow Types to Old Syntax Functions 318
A.1.24. “Address of” Operator 319
A.1.25. Unary Plus 319
A.1.26. Relational Operators 319
A.1.27. Assignment Compatibility 319
A.1.28. Declarations 319
A.1.28.1. Implementation Limits 319
A.1.28.2. Identifier Name Length 320
A.1.28.3. Diagnosing Empty Declarations 320
A.1.28.4. Restriction on Placement of Storage-Class Specifiers 320
A.1.28.5. Diagnosing Old-Style Function Declarations 320
A.1.28.6. Function Definitions Using typedef-names 320
A.1.28.7. Initialization 320
A.1.29. Bit-Field Initialization 320
A.1.30. The Preprocessor 320
A.1.30.1. White Space Appearing Before the # 321
A.1.30.2. The #define Directive and Macro Substitution 321
A.1.30.3. The #line Directive 321
A.1.30.4. The #error Directive 322
A.1.30.5. The #pragma builtins Directive 322
A.1.30.6. The #pragma dictionary Directive 322
A.1.30.7. The #pragma extern_model Directive 322
A.1.30.8. The #pragma linkage Directive (Alpha only) 322
A.1.30.9. The #pragma use_linkage Directive (Alpha only) 322
A.1.30.10. The #pragma message Directive 322
A.1.30.11. The #pragma module Directive 322
A.2. Features Affecting the VSI C Run-Time Library and Include Files 323
A.2.1. <stddef.h> 323
A.2.2. <ctype.h> 323
A.2.3. <fp_class.h> 323
A.2.4. <locale.h> 323
A.2.5. <math.h> 323
A.2.6. <signal.h> 323
A.2.7. <stdio.h> 323
A.2.8. <stdlib.h> 324
A.2.9. <string.h> 324
A.2.10. <time.h> 324
A.3. Unsupported Features 325
Appendix B. Common Pitfalls 327
Appendix C. Programming Tools 331
C.1. OpenVMS Debugger 331
C.1.1. Compiling and Linking to Prepare for Debugging 331

C.1.2. Starting and Terminating a Debugging Session 332
C.1.3. Notes on VSI C Support 332
C.1.3.1. Debugger Command-Line Options 333
C.1.3.2. Accessing Scalar Variables 333
C.1.3.3. Accessing Arrays 334
C.1.3.4. Accessing Character Strings 335
C.1.3.5. Accessing Structures and Unions 336
C.1.3.6. Sample Debugging Session 340
C.2. OpenVMS Text Processing Utility 343
C.3. Language-Sensitive Editor and the Source Code Analyzer 343
C.3.1. Preparing an SCA Library 344
C.3.2. Starting and Terminating an LSE or an SCA Session 345
C.3.3. Programming Language Placeholders and Tokens 345
C.3.4. Compiling Source Code 347
C.3.5. LSE Examples 347
C.3.5.1. Compilation Unit 347
C.3.5.2. Preprocessor Lines 348
C.4. CDD/Repository 348
C.4.1. Using CDD/Repository 349
C.4.2. Accessing CDD/Repository from VSI C Programs 349
C.4.3. Support for CDD/Repository Data Types 349
Appendix D. VSI C Compiler Messages 353
Appendix E. VSI C Limits 567
E.1. Contents of <float.h> 567
E.2. Contents of <limits.h> 573
VSI C Glossary 579

Preface

This manual provides reference information for using the VSI C language on OpenVMS systems. VSI C is an ANSI compliant C compiler for the OpenVMS operating system on VAX, Alpha, and Intel Itanium processors and for the UNIX operating system on Alpha processors. The shortened forms, OpenVMS IA-64 and I64, are also used throughout this manual.

VSI C is compliant with the International Standards Organization (ISO) C Standard (ISO 9899:1990[1992]), formerly the American National Standard for Information Systems-Programming Language C (document number: X3.159- 1989). By the use of command-line options, VSI C is compatible with older dialects of C, including common usage C (Kernighan and Ritchie C) and VAX C.

This manual is based on the ISO C Standard (ISO 9899:1990[1992]), formerly the ANSI X3J11 committee's standard for the C programming language (called the ANSI C standard in this manual). All library functions and language extensions to the ANSI C standard are also described. You may send comments or suggestions regarding this manual or any VSI C document by sending electronic mail to the following Internet address: `<docinfo@vmssoftware.com>`

1. About VSI

VMS Software, Inc. (VSI) is an independent software company licensed by Hewlett Packard Enterprise to develop and support the OpenVMS operating system.

2. Intended Audience

This guide is intended for experienced programmers who need to develop VSI C programs on OpenVMS systems, for users who need to know the difference between VSI C and other implementations, and for experienced C users who need to reference language information specific to OpenVMS systems. You should be familiar with one high-level language and should have some familiarity with the Digital Command Language (DCL). If you are not familiar with or need to reference information about the DCL, see *Chapter 1, "Developing VSI C Programs"*.

3. Document Structure

This guide has the following chapters and appendixes:

- *Chapter 1, "Developing VSI C Programs"* shows how to create, compile, link, and run a VSI C program.
- *Chapter 2, "Using OpenVMS Record Management Services"* describes VAX Record Management Services (RMS).
- *Chapter 3, "Using VSI C in the Common Language Environment"* describes interlanguage calling, and OpenVMS System Services, Run-Time Library (RTL) routines, and calling standard conventions.
- *Chapter 4, "Data Storage and Representation"* describes data storage and representation on OpenVMS systems.
- *Chapter 5, "Preprocessor Directives"* describes the preprocessor directives.

- *Chapter 6, "Predefined Macros and Built-In Functions"* describes the predefined macros and the built-in functions.
- *Appendix A, "Migrating from VAX C"* documents the features that distinguish VSI C for OpenVMS Systems from VAX C.
- *Appendix B, "Common Pitfalls"* describes common pitfalls when using VSI C.
- *Appendix C, "Programming Tools"* provides an overview of the OpenVMS Debugger, Text Processing Utility (TPU), Language-Sensitive Editor (LSE), Source Code Analyzer (SCA), and CDD/Repository.
- *Appendix D, "VSI C Compiler Messages"* lists VSI C compiler messages.
- *Appendix E, "VSI C Limits"* describes implementation-specific limits and parameters for VSI C on OpenVMS systems.
- The glossary provides an alphabetical listing of key terms.

4. Related Documents

You may find the following documents useful when programming in VSI C:

- *VSI C Reference Manual* [https://docs.vmssoftware.com/vsi-c-language-reference-manual/]—Provides language reference information for VSI C on OpenVMS systems.
- *VSI C Run-Time Library Reference Manual for OpenVMS Systems* [https://docs.vmssoftware.com/vsi-c-run-time-library-reference-manual-for-openvms-systems/]—Provides information on using the VSI C Run-Time Library (C RTL) functions and macros, and information about porting programs to and from other operating systems.
- The C Programming Language by Ritchie—Provides an excellent tutorial of the C language. Because VSI C contains features and enhancements to the standard C language, use the *VSI C User Manual* [https://docs.vmssoftware.com/vsi-c-user-s-guide-for-openvms-systems/] and the *VSI C Reference Manual* [https://docs.vmssoftware.com/vsi-c-language-reference-manual/] as the reference books for the full description of VSI C.
- *VSI OpenVMS Calling Standard*—Describes the concepts used by all OpenVMS languages to invoke routines and pass data between them. It also describes the differences between the OpenVMS VAX, Alpha, and I64 parameter-passing mechanisms.

5. OpenVMS Documentation

The full VSI OpenVMS documentation set can be found on the VMS Software Documentation webpage at https://docs.vmssoftware.com.

6. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: <docinfo@vmssoftware.com>. Users who have VSI OpenVMS support contracts through VSI can contact <support@vmssoftware.com> for help with this product.

7. Platform Labels

A *platform* is a combination of operating system and hardware that provides a distinct environment. This guide contains information applicable to the VSI OpenVMS operating system on VAX, Alpha, and Intel Itanium processors.

The information in this guide applies to all of these processors, except when specifically labeled as follows:

Label	**Explanation**
(VAX only)	Specific to a VAX processor running the OpenVMS operating system.
(Alpha only)	Specific to an Alpha processor running the OpenVMS operating system.
(I64 only)	Specific to an Intel Itanium processor running the OpenVMS operating system. On this platform, the product name of the operating system is OpenVMS Industry Standard 64 (or its abbreviated forms, OpenVMS IA-64 or I64).

8. Typographical Conventions

The conventions found in the following table are used in this document.

Convention	**Meaning**
UPPERCASE TYPE	All uppercase letters in a command line indicate keywords that must be entered. You can enter them in either uppercase or lowercase. You can use the first three characters to abbreviate command keywords, or you can use the minimum unique abbreviation.
lowercase italics	Lowercase italics in command syntax or examples indicate variables for which either you or the system supplies a value.
[ ]	In examples showing VMS directory specifications, square brackets are a necessary part of the specification, [*directory-name*]. In a procedure, square brackets in an inquiry enclose the default response for the inquiry.
[Return]	Press the Return key.
Ctrl/*x*	While holding down the Ctrl key, press the key specified by *x*.
. . .	Vertical ellipses (dots) in examples represent data that has been omitted.

9. New and Changed Features

VSI C runs on OpenVMS Alpha and OpenVMS Industry Standard 64 systems. The compiler behaves much the same on both systems, with some differences, primarily in the support for `#pragma linkage`, built-in functions, default floating-point representation, and predefined macros. These differences are noted in the relevant sections of this manual.

Chapter 1. Developing VSI C Programs

This chapter describes the following information about developing VSI C programs on an OpenVMS system:

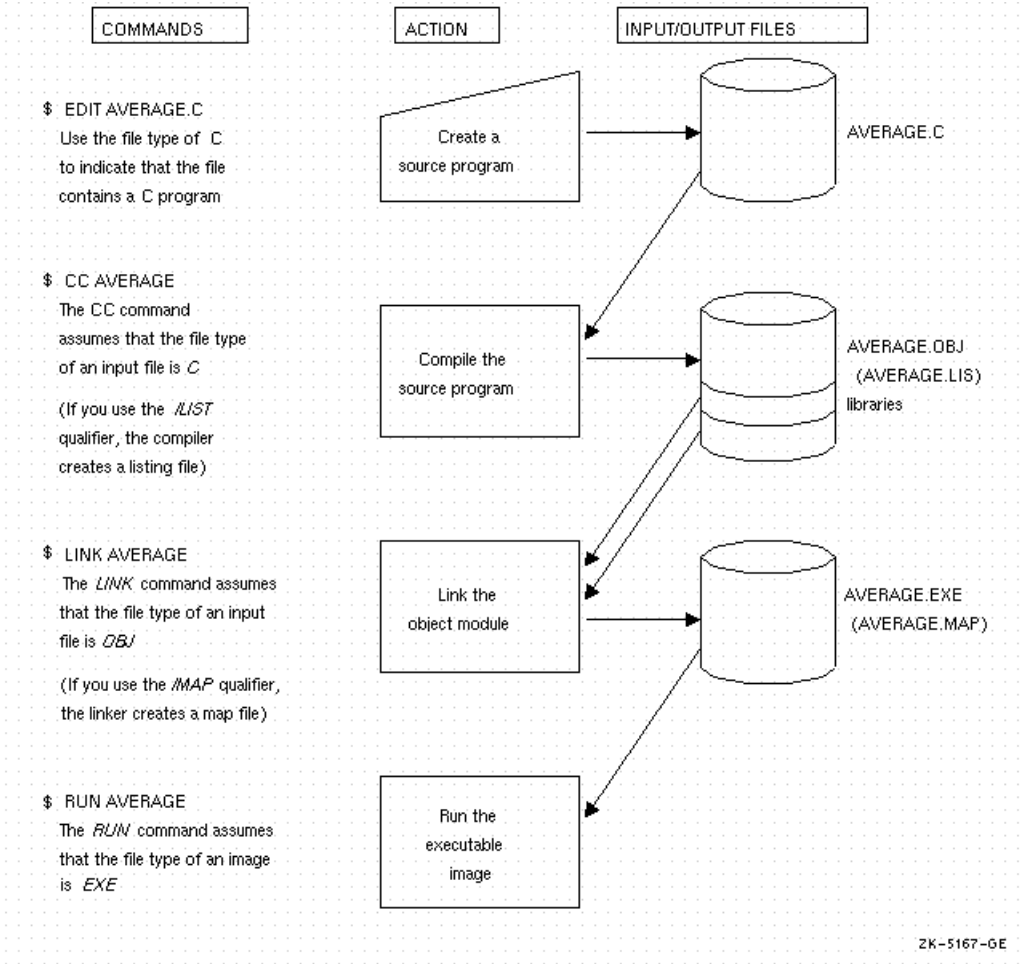
- Overview of the DIGITAL Command Language (DCL) commands used for program development (*Section 1.1, "DCL Commands for Program Development"*)
- Creating VSI C programs (*Section 1.2, "Creating a VSI C Program"*)
- Compiling VSI C programs (*Section 1.3, "Compiling a VSI C Program"*)
- Linking VSI C programs (*Section 1.4, "Linking a VSI C Program"*)
- Running VSI C programs (*Section 1.5, "Running a VSI C Program"*)
- Passing arguments to the `main` function (*Section 1.6, "Passing Arguments to the main Function"*)
- Using 64-bit addressing (*Section 1.7, "64-bit Addressing Support"*)

1.1. DCL Commands for Program Development

This section provides a brief overview of the DCL commands used for program development. The following sections provide more detailed information about these topics.

Figure 1.1, "DCL Commands for Developing Programs" shows the basic steps in VSI C program development.

Figure 1.1. DCL Commands for Developing Programs



To create a VSI C source program at DCL level, you must invoke a text editor. In *Figure 1.1, "DCL Commands for Developing Programs"*, the EDIT command invokes the default editor TPU (OpenVMS Text Processing Utility) to create the source program AVERAGE.C. You can use another editor, such as EDT or the *Language-Sensitive Editor* (LSE). (LSE is a product that must be purchased separately; see *Appendix C, "Programming Tools"* for more information.) A file type of C is used to indicate that you are creating a VSI C source program. C is the conventional file type for all VSI C source programs.

When you compile your program with the CC command, you do not have to specify the file type; by default, VSI C searches for files with a file type of C.

If your source program compiles successfully, the VSI C compiler creates an object file with the file type OBJ.

However, if the VSI C compiler detects errors in your source program, the system displays each error on your screen and then displays the DCL prompt. You can then reinvoke your text editor to correct each error.

You can specify command qualifiers on the CC command. Command qualifiers cause the VSI C compiler to perform additional actions. In the following example, the /LIST qualifier causes the VSI C compiler to produce the listing file AVERAGE.LIS:

```
$ CC/LIST  AVERAGE
```

For a complete description of all CC command qualifiers, see *Section 1.3.4, "CC Command Qualifiers"*.

After your program has compiled successfully, invoke the OpenVMS Linker to create an executable image file. For example:

```
$ LINK AVERAGE
```

The linker uses the object file produced by VSI C as input to produce an executable image file as output. (The executable image is a file containing program code that can be run on the system.)

You can specify command qualifiers with the DCL command LINK. For a complete list and explanation of all the command qualifiers available with the LINK command, see *Section 1.4.2, "LINK Command Qualifiers"*.

After producing the executable image file, use the RUN command to execute your program.

1.2. Creating a VSI C Program

To create and modify a VSI C program, you must invoke a text editor. The OpenVMS system provides you with two text editors: EDT and the OpenVMS Text Processing Utility (TPU). The following section discusses TPU. See the *OpenVMS EDT Reference Manual* for more information on EDT.

1.2.1. Using TPU

TPU is a high-performance, programmable utility. It provides two editing interfaces: the Extensible VAX Editor (EVE), described in the following section, and the TPU EDT Keypad Emulator. You can also create your own interfaces.

Like EDT, TPU provides you with an online help facility that you can access during your editing session. When you invoke TPU to create a file, a journal file is automatically created. You can use this journal file to recover your edits if the system fails during an editing session. To recover your edits, enter the EVE/RECOVER command.

Unlike EDT, TPU provides multiple windows. This feature allows you to view two files on your screen at the same time.

1.2.2. The EVE Interface to TPU

EVE is an interactive text editor that allows you to execute common editing functions using the EVE keypad or to execute more advanced functions by entering commands on the EVE command line. The following command line invokes the EVE editor and creates the file PROG_1.C:

```
$ EDIT/TPU PROG_1.C
```

You can define a global symbol for the EDIT/TPU command by placing a symbol definition in your LOGIN.COM file. For example:

```
$ EVE == "EDIT/TPU"
```

After this command line is executed, you can type EVE at the DCL prompt followed by the name of the file you want to modify or create.

1.3. Compiling a VSI C Program

The VSI C compiler performs the following functions:

- Detects errors in your source program
- Displays each error on your screen or writes the errors to a file
- Generates machine-language instructions from the source statements
- Groups these machine-language instructions into an object module for the linker

The following sections discuss the CC command and its qualifiers.

1.3.1. The CC Command

To invoke the VSI C compiler, enter the CC command at the DCL prompt ($). The CC command has the following format:

```
CC[/qualifier...][ file-spec [/qualifier...]],...
```

Note

(VAX only) This note applies to OpenVMS VAX systems that have both VSI C and VAX C installed.

The CC command is used to invoke either the VAX C or VSI C compiler. If the VSI C installation procedure detects that your system already has a VAX C compiler installed on it, the installer is given the option to specify which compiler gets invoked by default whenever the CC command verb is used. To invoke the compiler that is not the default, use the CC command with the appropriate qualifier: CC/DECC for the VSI C compiler, or CC/VAXC for the VAX C compiler. Where the CC command appears in examples in this manual, CC/DECC is assumed to be the default.

/qualifier

An action to be performed by the compiler on all files or specific files listed. When a qualifier appears directly after the CC command, it affects all the files listed. When a qualifier appears after a file specification, it affects only the file that immediately precedes it. However, when files are concatenated, these rules do not apply.

file-spec

An input source file that contains the program or module to be compiled. You are not required to specify a file type if you give your file a .C file extension; the VSI C compiler adopts the default file type C.

You can include more than one file specification on the same command line by separating the file specifications with either a comma (,) or a plus sign (+). If you separate the file specifications with commas, you can control which source files are affected by each qualifier. In the following example, the VSI C compiler creates an object file for each source file but creates only a listing file for the source files PROG_1 and PROG_3:

```
$ CC /LIST PROG_1, PROG_2/NOLIST, PROG_3
```

If you separate file specifications with plus signs, the VSI C compiler concatenates each of the specified source files and creates one object file and one listing file. In the following example, only one object file is created, PROG_1.OBJ, and only one listing file is created, PROG_1.LIS. Both of these files are named after the first source file in the list, but contain all three modules.

```
$ CC PROG_1 + PROG_2/LIST + PROG_3
```

Any qualifiers specified for a single file within a list of files separated with plus signs affect all the files in the list. See the description of the /PLUS_LIST_OPTIMIZE qualifier for its affect on file concatenation.

Note

Concatenating source files without using the /PLUS_LIST_OPTIMIZE qualifier is not recommended because potential conflicts in the name space of declared objects can result in compilation errors or incorrect run-time behavior.

A more common use of plus-list concatenation is for specifying text libraries. You can specify the name of a text library on the CC command line to compile a source program. A text library is a file that contains text organized into modules indexed by a table. Text libraries have a .TLB default file extension. In the following example, text libraries A.TLB and B.TLB are made available for searching for text library modules during the compilation of source file TEST.C:

```
$ CC TEST.C + A.TLB/LIB + B.TLB/LIB
```

1.3.1.1. Including Header Files

Header files are pieces of source code that typically contain declarations shared among C programs. A header file often declares a set of related functions, as well as defining any types and macros needed for their use.

To make the contents of a header file available to your program, include the header file using the `#include` preprocessor directive.

The `#include` directive has three forms. Two of the forms are defined by the C standard and are portable:

- Inclusion using angle brackets to delimit the file to be included:

  `#include <`*`file-spec`*`>`

- Inclusion using quotation marks to delimit the file to be included:

  `#include "`*`file-spec`*`"`

The third form is the text-module form. It is specific to OpenVMS systems and is not portable. See *Section 5.2.3, "Inclusion of Text Modules"* for more information on the text-module form of inclusion.

The form of the `#include` directive used determines where the compiler will look to find the file to be included. Generally, the compiler looks in the following places, in the order listed:

1. Places named on the command line with the /INCLUDE_DIRECTORY qualifier or the /LIBRARY qualifier

2. Places identified through logical names, such as DECC$USER_INCLUDE, DECC$SYSTEM_INCLUDE, DECC$LIBRARY_INCLUDE, and DECC$TEXT_LIBRARY

3. System-defined places such as the SYS$COMMON:[DECC$LIB.INCLUDE.*] directory and the SYS$LIBRARY:DECC$RTLDEF.TLB and SYS$LIBRARY:SYS$STARLET_C.TLB text libraries

You can use the UNUSED message group described in the `#pragma message` description in *Section 5.4.14, "#pragma message Directive"* to enable messages that report apparently unnecessary `#include` files (and CDD records). Unlike any other messages, these messages must be enabled on the command line (/WARNINGS=ENABLE=UNUSED), rather than with `#pragma message`, to be effective.

The VSI C preprocessor is usually able to determine if a particular `#include` file that has already been processed once was guarded by the conventional sequence: `#ifndef FILE_SEEN, #define FILE_SEEN, #endif`.

When the compiler detects this pattern of use the first time a particular file is included, it remembers that fact as well as the name of the macro. The next time the same file is included, the compiler checks to see if the "FILE_SEEN" macro is still defined and, if so, it does not reopen and reread the file. Note that if the initial test is in the form `#if !defined` instead of `#ifndef`, then the pattern is not recognized. In a listing file, `#include` directives that are skipped because of this processing are marked with an "X" just as if the `#include` line itself were excluded.

See the /INCLUDE_DIRECTORY qualifier in *Section 1.3.4, "CC Command Qualifiers"* for a more complete description of the search-order rules that VSI C uses to locate included files.

See the *VSI C Run-Time Library Reference Manual for OpenVMS Systems* [https://docs.vmssoftware.com/vsi-c-run-time-library-reference-manual-for-openvms-systems/] for information on the header files required to use VSI C Run-Time Library (C RTL) functions and macros.

1.3.1.2. Listing Header Files

To list the names of system header files, use the following commands:

```
$ LIBRARY/LIST SYS$LIBRARY:SYS$STARLET_C.TLB
(OpenVMS Version 7.1 and higher)

$ LIBRARY/LIST SYS$LIBRARY:DECC$RTLDEF.TLB
$ DIR SYS$COMMON:[DECC$LIB.REFERENCE.SYS$STARLET_C]*.H;
$ DIR SYS$COMMON:[DECC$LIB.REFERENCE.DECC$RTLDEF]*.H;
$ DIR SYS$LIBRARY:*.H;
```

These commands list, respectively:

- The names of the text-module header files for the OpenVMS system interfaces
- The names of the text-module header files for the VSI C language interfaces
- *.h header files for the OpenVMS system interfaces
- *.h header files for the VSI C language interfaces
- *.h header files for layered products and other applications

Note

The SYS$COMMON:[DECC$LIB.REFERENCE.DECC$RTLDEF] and SYS$COMMON:[DECC$LIB.REFERENCE.SYS$STARLET_C] directories are only reference areas for your viewing. They are created during the compiler installation from the content of the text libraries. By default, the compiler searches only the text library files for headers; it does not search these reference directories.

Be aware that OpenVMS VAX operating systems prior to Version 7.1 do not have a file named SYS$LIBRARY:SYS$STARLET_C.TLB. For these older versions of the operating system, the STARLET header files are generated during VSI C installation and placed in SYS$LIBRARY:DECC$RTLDEF.TLB and also in both SYS$COMMON:[DECC$LIB.REFERENCE.DECC$RTLDEF] and SYS$COMMON:[DECC$LIB.REFERENCE.SYS$STARLET_C].

1.3.2. Compilation Modes

VSI C has two complementary qualifiers that control which dialect of C is to be recognized by the compiler, and which messages are generated:

- The /STANDARD qualifier controls what language features and extensions are recognized by the compiler.
- The /[NO]WARNINGS qualifier enables or disables the generation of warning and/or informational messages.

The /STANDARD qualifier causes the compiler to issue only those warnings appropriate for the dialect of C being compiled. For example, VAX C compatibility mode (/STANDARD=VAXC) does not issue warnings against VAX C extensions, while ANSI C mode does.

To generate a list of all messages that are in effect at the start of compilation, specify /LIST/SHOW=MESSAGES. For each message, the identifier, severity, and message text are shown. To also show the message description and user action for each message listed, specify /LIST/SHOW=MESSAGES/WARN=VERBOSE.

The VSI C compiler for OpenVMS systems provides several dialects of C, which are controlled by the /STANDARD qualifier:

- Strict ANSI C: Only the ANSI C Standard 89 (C89) language dialect is recognized. This mode is enabled by specifying /STANDARD=ANSI89 on the CC command line.

  /STANDARD=ANSI89 issues all diagnostics required by the ANSI C standard as well as a number of optional diagnostics that help detect source code constructs that are not portable under the C89 standard. Digraph recognition from the 1994 Amendment is also supported in this mode.

  You can use /STANDARD=ANSI89 with /[NO]WARNINGS to control issuance of informational or warning messages. However, since the compiler does not recognize many VAX C or common C extensions when in strict ANSI mode (for example, VAX C keywords not beginning with two underscores), many of the messages normally associated with flagging VAX C and common C extensions are not produced.

- Strict C99: Only the ISO C99 dialect is recognized. This mode is enabled by specifying /STANDARD=C99 on the CC command line.

  /STANDARD=C99 accepts just the C99 language without extensions, and diagnoses violations of the C99 standard. Because C99 is a superset of Amendment 1 to the C89 standard, and the default mode of RELAXED is a superset of C99, the __STDC_VERSION__ macro is now defined with the C99-specified value of 199901L.

  Only when the ISOC94 keyword is added to the strict ANSI89, MIA, or COMMON modes does the __STDC_VERSION__ macro take on the Amendment 1 value of 199409L (In the absence of the ISOC94 keyword, the ANSI89, MIA, and COMMON modes do not define the macro at all.)

Note

/STANDARD=C99 is not fully supported on VAX systems. Specifying /STANDARD=C99 on OpenVMS VAX systems produces a warning and puts the compiler into /STANDARD=RELAXED mode.

- Latest C standard dialect. /STANDARD=LATEST is currently equivalent to /STANDARD=C99, but is subject to change when newer versions of the C standard are released.
- Relaxed: This is the default mode on OpenVMS systems, and is specified by /NOSTANDARD or /STANDARD=RELAXED on the CC command line. The /STANDARD=RELAXED mode accepts C89 and C99 features, as well as nearly all language extensions (such as additional VSI C keywords and predefined macros that do not begin with an underscore). It excludes only K&R (COMMON mode), VAX C, and Microsoft features that conflict with standard C. The purpose of the /STANDARD=RELAXED mode is to support everything from the most current C standard, in addition to all extensions that do not specify different semantics for the same constructs.
- Microsoft compatibility: This mode interprets source programs according to certain language rules followed by the C compiler provided with the Microsoft Visual C++ compiler product. This mode is enabled by specifying /STANDARD=MS on the CC command line. See *Section 1.3.3, "Microsoft Compatibility Compilation Mode"* for more information about Microsoft compatibility mode.
- ISO C 94: This mode is enabled by specifying /STANDARD=ISOC94. It can be specified alone or with any other /STANDARD option except VAXC. If it is specified alone, the default major mode is RELAXED.

  Specifying /STANDARD=ISOC94 enables digraph processing and defines the predefined macro __STDC_VERSION__=199409L, as specified by Amendment 1 to the C89 standard.
- VAX C compatibility: This mode is enabled by specifying /STANDARD=VAXC. It allows the same language as the C standard, but also supports VAX C extensions that are incompatible with the C standard and that change the language semantics. This mode provides compatibility for programs that depend on old VAX C behavior.
- Portable: This mode is enabled by specifying /STANDARD=PORTABLE. It places the compiler in RELAXED mode and enables the issuance of diagnostics that warn about any nonportable usages encountered.

  /STANDARD=PORTABLE is supported for VAX C compatibility only. It is equivalent to the recommended combination of qualifiers /STANDARD=RELAXED/ WARNINGS=ENABLE=PORTABLE.
- Common usage C: This mode is enabled by specifying /STANDARD=COMMON. It enforces K & R programming style; that is, compatibility with older UNIX compilers such as `pcc` and `gcc`. This mode is close to a subset of /STANDARD=VAXC mode.
- MIA conformance: This mode is enabled by specifying /STANDARD=MIA. This is strict ANSI C with some differences required by the Multivendor Integration Architecture (MIA) standard. Compiling a program with /STANDARD=MIA sets the `__MIA` predefined macro to 1.

With one exception, the /STANDARD qualifier options are mutually exclusive. Do not combine them. The exception is that you can specify /STANDARD=ISOC94 with any other option except VAXC.

VSI C modules compiled in different modes can be linked and executed together.

The /STANDARD qualifier is further described in *Section 1.3.4, "CC Command Qualifiers"*.

Also see the `__HIDE_FORBIDDEN_NAMES` predefined macro (*Section 6.1.7, "The __HIDE_FORBIDDEN_NAMES Macro"*).

1.3.3. Microsoft Compatibility Compilation Mode

The /STANDARD=MS qualifier instructs the VSI C compiler to interpret your source code according to certain language rules followed by the C compiler provided with the Microsoft Visual C++ compiler product. However, compatibility with this implementation is not complete. The following sections describe the compatibility situations that VSI C recognizes. In most cases, these situations consist of relaxing a standard behavior and suppressing a diagnostic message.

1.3.3.1. Unnamed Nested struct or union Members

Allow a declaration of a structure with no name within another structure. You can reference all members of the inner structure as members of the named outer structure. This is similar to the C++ treatment of nested unions lacking a name, but extended to both structures and unions. A similar capability is provided by the VAX C `variant_struct` and `variant_union` types.

For example:

```
struct{
  struct{
      int a;
      int b;
  };  /*No name here */
  int c;
}d;  /* d.a, d.b, and d.c are valid member names. */
```

1.3.3.2. Block Scope Declaration of static Functions

Allow a `static` function declaration in block scope (that is, inside another function).

For example:

```
f(){
   static int a(int b);
}
```

1.3.3.3. Treat &* as Having No Effect

Standard C does not allow the `&` operator to produce an lvalue expression. The Microsoft relaxation allows `&` to produce an lvalue in certain cases.

For example:

```
int *a, *b;

f() {

  &*a=b;

}
```

1.3.3.4. char is Not Treated as a Unique Type

Treat the `char` type as either `signed char` or , depending on the default in effect.

For example, a pointer to `char` can be assigned to a pointer to `signed char`, assuming the command-line default of /NOUNSIGNED_CHAR:

```
signed char *a;
char *b;

f() {
b=a;
}
```

1.3.3.5. Double Semicolons in Declarations

Suppress warning messages for declarations that contain two semicolons. (That is, allow completely empty declarations at file scope.)

For example:

```
int a;;
```

1.3.3.6. Declaration without a Type

Suppress warning messages for declarations that contain a variable name but no type.

For example:

```
b;
```

1.3.3.7. Enumerators in an Enumeration Declaration

Ignore any extra comma at the end of the last enumerator in an enumeration declaration.

For example:

```
enum E {a, b, c,};    /*  Ignore the comma after "c". */
```

1.3.3.8. Useless Typedefs

Allow `typedefs` that have a type specifier but no identifier name declaring the new type.

For example:

```
typedef struct { int a; };
```

1.3.3.9. Unrecognized Pragmas Accepted

Suppress warning messages when one of the following unsupported Microsoft pragmas is encountered:

```
#pragma code_seg
#pragma warning
```

1.3.4. CC Command Qualifiers

The following list shows all the command qualifiers and their defaults available with the CC command. A description of each qualifier follows the list.

You can place command qualifiers either on the CC command line itself or on individual file specifications (with the exception of the /LIBRARY qualifier). If placed on a file specification, the

qualifier affects only the compilation of the specified source file and all subsequent source files in the compilation unit. If placed on the CC command line, the qualifier affects all source files in all compilation units unless it is overridden by a qualifier on an individual file specification.

Command Qualifiers	**Default**
/ACCEPT=(option[,option])	See text.
/[NO]ANALYSIS_DATA[=file-spec]	/NOANALYSIS_DATA
/[NO]ANNOTATIONS[=(option,…)]	/NOANNOTATIONS
/[NO]ANSI_ALIAS	See text.
/ARCHITECTURE=option	/ARCHITECTURE=GENERIC
/ASSUME=(option[,…])	See text.
/[NO]CHECK[=(option,…)]	/NOCHECK
/[NO]COMMENTS=option	See text.
/[NO]CROSS_REFERENCE	/NOCROSS_REFERENCE
/[NO]DEBUG[=(option[,…])]	/DEBUG=(TRACEBACK,NOSYMBOLS) /DEBUG=(TRACEBACK,NOINLINE, NOSYMBOLS) (VAX only)
/DECC	See text.
/[NO]DEFINE=(identifier[=definition][,…])	/NODEFINE
/[NO]DIAGNOSTICS[=file-spec]	/NODIAGNOSTICS
/ENDIAN=option	/ENDIAN=LITTLE
/[NO]ERROR_LIMIT[=n]	/ERROR_LIMIT=30
/EXTERN_MODEL=option	/EXTERN_MODEL=RELAXED_REFDEF
/[NO]FIRST_INCLUDE=(file[,…])	/NOFIRST_INCLUDE
/FLOAT=option	/FLOAT=G_FLOAT (Alpha only) /FLOAT=IEEE_FLOAT (I64 only) /FLOAT=D_FLOAT (VAX only)
/GRANULARITY=option	/GRANULARITY=QUADWORD
/[NO]INCLUDE_DIRECTORY=(pathname[,…])	/NOINCLUDE_DIRECTORY
/IEEE_MODE[=option]	/IEEE_MODE=FAST (Alpha only) /IEEE_MODE=DENORM_RESULTS (I64 only)
/L_DOUBLE_SIZE=option	/L_DOUBLE_SIZE=128
/LIBRARY	See text.
/[NO]LINE_DIRECTIVES	/LINE_DIRECTIVES
/[NO]LIST[=file-spec]	/NOLIST (interactive mode) /LIST (batch mode)

Command Qualifiers	**Default**
/[NO]MACHINE_CODE[=option]	/NOMACHINE_CODE
/[NO]MAIN=POSIX_EXIT	/NOMAIN
/[NO]MEMBER_ALIGNMENT	/MEMBER_ALIGNMENT /NOMEMBER_ALIGNMENT (VAX only)
/[NO]MMS_DEPENDENCIES=option	/NOMMS_DEPENDENCIES
/NAMES=(option1,option2)	/NAMES=UPPERCASE,TRUNCATED
/NESTED_INCLUDE_DIRECTORY[=option]	/NESTED_INCLUDE_DIRECTORY
	=INCLUDE_FILE
/[NO]OBJECT[=file-spec]	/OBJECT
/[NO]OPTIMIZE[=(option[,…])]	/OPTIMIZE
/PDSC_MASK=option	See text.
/[NO]PLUS_LIST_OPTIMIZE	/NOPLUS_LIST_OPTIMIZE
/[NO]POINTER_SIZE=option	/NOPOINTER_SIZE
/PRECISION[=option]	See text.
/[NO]PREFIX_LIBRARY_ ENTRIES[=(option[,…])]	See text.
/[NO]PREPROCESS_ONLY[=filename]	/NOPREPROCESS_ONLY
/[NO]PROTOTYPES[=(option[,…])]	/NOPROTOTYPES
/PSECT_MODEL=[NO]MULTILANGUAGE	/NOMULTILANGUAGE
/REENTRANCY=option	/REENTRANCY=TOLERANT
/REPOSITORY=option	/See text.
/ROUNDING_MODE=option	/ROUNDING_MODE=NEAREST
/[NO]SHARE_GLOBALS	/NOSHARE_GLOBALS
/SHOW[=(option[,…])]	/SHOW=(NOBRIEF, NOCROSS_REFERENCE, NODICTIONARY, NOEXPANSION, NOINCLUDE, NOINTERMEDIATE, NOMESSAGE, NOSTATISTICS, NOSYMBOLS, NOTRANSLATION, SOURCE, TERMINAL)
/[NO]STANDARD[=(option[,…])]	/NOSTANDARD (equivalent to /STANDARD=RELAXED)
/[NO]TIE	/NOTIE
/[NO]UNDEFINE=(identifier[,…])	/NOUNDEFINE
/[NO]UNSIGNED_CHAR	/NOUNSIGNED_CHAR
/VAXC (VAX only)	See text.
/[NO]VERSION	/NOVERSION
/[NO]WARNINGS[=(option[,…])]	/WARNINGS

/ACCEPT=(option[,option])

Allows the compiler to accept C language syntax that it might not normally accept.

VSI C accepts slightly different syntax depending upon the compilation mode specified with the /STANDARD qualifier. The /ACCEPT qualifier can fine tune the language syntax accepted by each /STANDARD mode.

The following qualifier options can be specified:

Table 1.1. /ACCEPT Qualifier Options

Option	Usage
[NO]C99_KEYWORDS	Controls whether or not the C99 Standard keywords `inline` and `restrict` (which are in the C89 namespace for user identifiers) are accepted without double leading underscores. The spelling with two leading underscores (`__inline`, `__restrict`) is in the namespace reserved to the compiler implementation and is always recognized as a keyword regardless of this option.
[NO]GCCINLINE	The gcc compiler implements an `inline` function qualifier for functions with external linkage that gives similar capabilities as the C99 `extern inline` feature for functions, but the usage details are somewhat different: the combination of `extern` and `inline` keywords makes an inline definition, instead of the exclusive use of the `inline` keyword without the `extern` keyword. This option controls which variation of the feature is implemented. The default in all compiler modes is NOGCCINLINE.
[NO]RESTRICT_KEYWORD	Controls whether or not the compiler recognizes the C99 standard `restrict` keyword regardless of the /STANDARD mode used. This only affects recognition of the spelling of the keyword as proposed for inclusion in the C99 standard. The spelling with two leading underscores, `__restrict`, is in the namespace reserved to the compiler implementation and is always recognized as a keyword regardless of this option. Note that [NO]RESTRICT_KEYWORD is a subset of [NO]C99_KEYWORDS. They have the same compiler-mode defaults.
[NO]TRIGRAPHS	Turns trigraph processing on or off. In COMMON and VAXC modes, trigraphs are disabled by default. In all other modes, they are enabled by default.
[NO]VAXC_KEYWORDS	Controls whether or not the compiler recognizes the VAX C keywords (such as "readonly") regardless of the /STANDARD mode used.

The default values are based upon the settings of the /STANDARD qualifier:

- For /STANDARD=RELAXED, the default is:

  /ACCEPT=(VAXC_KEYWORDS,C99_KEYWORDS, NOGCCINLINE,TRIGRAPHS)

- For /STANDARD=VAXC, the default is:

  /ACCEPT=(VAXC_KEYWORDS,NOC99_KEYWORDS, NOGCCINLINE,NOTRIGRAPHS)

- For /STANDARD=COMMON, the default is:

  /ACCEPT=(NOVAXC_KEYWORDS,NOC99_KEYWORDS, NOGCCINLINE,NOTRIGRAPHS)

- In all other modes, the default is:

  /ACCEPT=(NOVAXC_KEYWORDS,NOC99_KEYWORDS, NOGCCINLINE,TRIGRAPHS)

/[NO]ANALYSIS_DATA[=file-spec]

Generates a file of source-code analysis information. The default file name is the file name of the primary source file; the default file type is .ANA. The .ANA file is reserved for use with VSI layered products. The default is /NOANALYSIS_DATA. For more information, see *Appendix C, "Programming Tools"*.

/[NO]ANNOTATIONS[=option]

Controls whether or not the source listing file is annotated with indications of specific optimizations performed or, in some cases, not performed. These annotations can be helpful in understanding the optimization process.

If annotations are requested (and the /LISTING qualifier appears on the command line), the source listing section is shifted to the right and annotation numbers are added to the left of source lines. These numbers refer to brief descriptions that appear later in the source listing file.

Select one or more of the /ANNOTATIONS qualifier options shown in *Table 1.2, "/ANNOTATIONS Qualifier Options"*.

Table 1.2. /ANNOTATIONS Qualifier Options

Option	Usage
ALL	Selects all annotations. This output can be quite verbose because it includes detailed output for all annotations. For more concise output for each kind of annotation, use /ANNOTATIONS=(ALL,NODETAIL), or just /ANNOTATIONS with no qualifier options.
[NO]CODE	Annotates the machine-code listing with descriptions of special instructions used for prefetching, alignment, and so on. The /MACHINE_CODE qualifier must also be specified for /ANNOTATION=CODE to have any visible effect.
[NO]DETAIL	Provides additional level of annotation detail, where available.
[NO]FEEDBACK	Indicates use of profile-directed feedback optimizations. Feedback optimizations are not implemented on OpenVMS systems, so this keyword has no visible effect.
[NO]INLINING	Indicates where code for a called procedure was expanded inline.
[NO]LOOP_TRANSFORMS	Indicates optimizations such as loop reordering and code hoisting.

Option	Usage
[NO]LOOP_UNROLLING	Indicates where advanced loop nest optimizations have been applied to improve cache performance (unroll and jam, loop fusion, loop interchange, and so on).
[NO]PREFETCHING	Indicates where special instructions were used to reduce memory latency.
[NO]SHRINKWRAPPING	Indicates removal of code establishing routine context when it is not needed.
[NO]SOFTWARE_PIPELINING	Indicates where loops have been scheduled to hide functional unit latency.
[NO]TAIL_CALLS	Indicates an optimization where a call from routine A to B can be replaced by a jump.
[NO]TAIL_RECURSION	Indicates an optimization that eliminates unnecessary routine context for a recursive call.
NONE	Same as /NOANNOTATIONS.

The default is /NOANNOTATIONS.

Specifying /ANNOTATIONS with no keywords is the same as
specifying /ANNOTATIONS=(ALL,NODETAIL).

/[NO]ANSI_ALIAS

Directs the compiler to assume the standard C aliasing rules. By so doing, the compiler has the freedom to generate better optimized code.

The aliasing rules referred to are explained in the C Standard, reprinted as follows:

An object shall have its stored value accessed only by an lvalue that has one of the following types:

- the declared type of the object,
- a qualified version of the declared type of the object,
- a type that is the signed or unsigned type corresponding to the declared type of the object,
- a type that is the signed or unsigned type corresponding to a qualified version of the declared type of the object,
- an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or
- a character type.

If your program does not access the same data through pointers of a different type (and for this purpose, signed and qualified versions of an otherwise same type are considered to be the same type), then assuming standard C aliasing rules allows the compiler to generate better optimized code.

If your program does access the same data through pointers of a different type (for example, by a "pointer to `int`" and a "pointer to `float`"), then you must not allow the compiler to assume standard C aliasing rules. Otherwise, incorrect code might be generated.

The default is /NOANSI_ALIAS for the /STANDARD=VAXC and /STANDARD=COMMON compiler modes. The default is /ANSI_ALIAS for all other modes.

/ARCHITECTURE

Determines the Alpha or Intel processor instruction set to be used by the compiler.
The /ARCHITECTURE qualifier uses the same keyword options (keywords) as the /OPTIMIZE=TUNE qualifier.

Where the /OPTIMIZE=TUNE qualifier is primarily used by certain higher-level optimizations for instruction scheduling purposes, the /ARCHITECTURE qualifier determines the type of code instructions generated for the program unit being compiled.

OpenVMS Version 7.1 and subsequent releases provide an operating system kernel that includes an instruction emulator. This emulator allows new instructions, not implemented on the host processor chip, to execute and produce correct results. Applications using emulated instructions will run correctly, but may incur significant software emulation overhead at runtime.

All Alpha processors implement a core set of instructions. Certain Alpha processor versions include additional instruction extensions.

Select one of the /ARCHITECTURE qualifier options shown in *Table 1.3, "/ARCHITECTURE Qualifier Options"*.

Table 1.3. /ARCHITECTURE Qualifier Options

Option	Usage
GENERIC	Generates code that is appropriate for all Alpha and Itanium processor generations. This is the default.
HOST	Generates code for the processor generation in use on the system being used for compilation. Running programs compiled with this option on other implementations of the Alpha or Itanium architecture may encounter instruction-emulation overhead.
EV4 (Alpha only)	Generates code for the 21064, 21064A, 21066, and 21068 implementations of the Alpha architecture. Running programs compiled with the EV4 option will run without instruction-emulation overhead on all Alpha processors.
EV5 (Alpha only)	Generates code for some 21164 chip implementations of the Alpha architecture that use only the base set of Alpha instructions (no extensions). Running programs compiled with the EV5 option will run without instruction-emulation overhead on all Alpha processors.
EV56 (Alpha only)	Generates code for some 21164 chip implementations that use the byte and word-manipulation instruction extensions of the Alpha architecture. Running programs compiled with the EV56 option might incur emulation overhead on EV4 and EV5 processors, but will still run correctly on OpenVMS Version 7.1 (or higher) systems.

Option	Usage
PCA56 (Alpha only)	Generates code for the 21164PC chip implementation that uses the byte- and word-manipulation instruction extensions and multimedia instruction extensions of the Alpha architecture. Running programs compiled with the PCA56 option might incur emulation overhead on EV4, EV5, and EV56 processors, but will still run correctly on OpenVMS Version 7.1 (or higher) systems.
EV6 (Alpha only)	Generates code for the first-generation 21264 implementation of the Alpha architecture.
EV67 (Alpha only)	Generates code for the second-generation 21264 implementation of the Alpha architecture.
ITANIUM2 (I64 only)	Generates code for the Intel Itanium 2 processor.

`/ASSUME=(option,...)`

Controls compiler assumptions. You can select one or more of the qualifier options described in *Table 1.4, "/ASSUME Qualifier Options"*.

Table 1.4. /ASSUME Qualifier Options

Option	Usage
[NO]ACCURACY_SENSITIVE	Specifies whether certain code transformations that affect floating-point operations are allowed. These changes may or may not affect the accuracy of the program's results.
[NO]ALIGNED_OBJECTS	Controls an optimization for dereferencing pointers.
[NO]CLEAN_PARAMETERS	Controls compiler assumptions about short-integer formal parameters.
[NO]EXACT_CDD_OFFSETS	Controls the alignment of Control Data Dictionary records.
[NO]HEADER_TYPE_DEFAULT	Controls whether or not the default file-type mechanism for header files is enabled.
[NO]MATH_ERRNO	Controls whether or not intrinsic code is generated for math functions that set the errno variable.
[NO]POINTERS_TO_GLOBALS	Controls whether or not the compiler can safely assume that global variables have not had their addresses taken in code that is not visible to the current compilation.
[NO]WEAK_VOLATILE	Affects the generation of code for assignments to objects that are less than or equal to 16 bits in size that have been declared as volatile.
[NO]WHOLE_PROGRAM	Asserts to the compiler that except for "well-behaved library routines," the whole program consists only of the single object module being produced by this compilation.
[NO]WRITABLE_STRING_ LITERALS	Stores string constants in a writable psect. Otherwise, such constants are placed in a nonwritable psect.

The following sections describe these options in greater detail.

[NO]ACCURACY_SENSITIVE

The default is ACCURACY_SENSITIVE.

If you specify NOACCURACY_SENSITIVE, the compiler is free to reorder floating-point operations based on algebraic identities (inverses, associativity, and distribution). This allows the compiler to move divide operations outside of loops, which improves performance.

The default, ACCURACY_SENSITIVE, directs the compiler to use only certain scalar rules for calculations. This setting can prevent some optimizations.

If you use the /ASSUME=NOACCURACY_SENSITIVE qualifier, VSI C might reorder code (based on algebraic identities) to improve performance. The results can be different from the default (/ASSUME=ACCURACY_SENSITIVE) because of how the intermediate results are rounded. However, the NOACCURACY_SENSITIVE results are not categorically less accurate than those gained by the default.

[NO]ALIGNED_OBJECTS

The default is /ASSUME=ALIGNED_OBJECTS.

On OpenVMS Alpha and IA-64 systems, dereferencing a pointer to a longword- or quadword-aligned object is more efficient than dereferencing a pointer to a byte- or word-aligned object. Therefore, the compiler can generate more optimized code if it makes the assumption that a pointer object of an aligned pointer type does point to an aligned object.

Since the compiler determines the alignment of the dereferenced object from the type of the pointer, and the program is allowed to compute a pointer that references an unaligned object (even though the pointer type indicates that it references an aligned object), the compiler must assume that the dereferenced object's alignment matches or exceeds the alignment indicated by the pointer type. Specifying /ASSUME=ALIGNED_OBJECTS (the default) allows the compiler to make such an assumption. With this assumption made, the compiler can generate more efficient code for pointer dereferences of aligned pointer types.

To prevent the compiler from assuming the pointer type's alignment for objects that it points to, use the /ASSUME=NOALIGNED_OBJECTS qualifier.

Before deciding whether to specify /ASSUME=NOALIGNED_OBJECTS or /ASSUME=ALIGNED_OBJECTS, you need to know what programming practices will affect your decision.

The compiler assumes that pointers point to objects that are aligned at least as much as the alignment of the pointer type. For example:

- A pointer of type `short` points to objects that are at least `short`-aligned.
- A pointer of type `int` points to objects that are at least `int`-aligned.
- A pointer of type `struct foo` points to objects that have an alignment of `struct foo` (that is, the alignment of the strictest member alignment, or byte alignment if you have specified `#pragma nomember_alignment` for `struct foo`).

If your module breaks this rule, your program will suffer alignment faults at runtime that can seriously degrade performance. If you can identify the places in your code where the rule is broken, use the `__unaligned` type qualifier. Otherwise, the /ASSUME=NOALIGNED_OBJECTS qualifier effectively treats all dereferences as if they were unaligned.

On OpenVMS Alpha and IA-64 systems, VSI C aligns all nonmember declarations on natural boundaries, so by default all objects do comply with the previous assumption. Also, the standard library routine `malloc` on OpenVMS systems returns quadword-aligned heap memory.

A program can violate the previous assumption in any of the following ways:

- By explicitly specifying a lesser alignment for an object than the pointer type's alignment
- By casting a pointer to a pointer type of stricter alignment
- By enclosing a member-aligned object inside a nonmember-aligned object

The following example explicitly specifies a lesser alignment for an object than the pointer type's alignment, which occurs when the address of an unaligned `int` member of a `struct` with `#pragma nomember_alignment` is used in a pointer dereference:

```
#pragma nomember_alignment
struct foo {
    char C;
    int i; /* i is unaligned because of char C */
};

struct foo st;
int        *i_p;

i_p = &st.i;

... *i_p ...    /* An expression containing a dereferenced i_p */
```

This example casts a pointer to a pointer type with stricter alignment:

```
int        *i_p;
char       *c_p;

.......
.......

i_p = (int *)c_p;

... *i_p ...    /* An expression containing a dereferenced i_p */
```

The following example encloses a member-aligned object inside a nonmember-aligned object:

```
#pragma member_alignment
struct inside {
    int i;  /* this type asserts that its objects have at least
               longword alignment (int is a longword)... */
};

#pragma nomember_alignment
struct outside {
    char C;
    struct inside s; /* ...but foo_ptr -> s is only byte-aligned! */
} *foo_ptr;
```

The expression `foo_ptr -> s` has a type whose alignment is explicitly specified to be longword (because longword is the strictest alignment of the structure's members), but the expression type is only guaranteed to be byte-aligned.

Also note that just as the pointer type information can direct the compiler to generate the appropriate code to dereference the pointer (code that does not cause alignment faults), it can also direct the compiler to generate even better code if it indicates that the object is at least longword-aligned.

[NO]CLEAN_PARAMETERS

The default is /ASSUME=CLEAN_PARAMETERS.

The OpenVMS Alpha and I64 Calling Standards require integers less than 64 bits long that are passed by value to have their upper bits either zeroed or sign-extended to make full 64-bit values. These are referred to as clean parameters. Some old code does not follow this convention. This can cause problems if the called program assumes that the caller followed the Calling Standard by passing only clean parameters.

Specifying /ASSUME=NOCLEAN_PARAMETERS allows a program to be called by old code that might pass unclean integer parameters. It directs the compiler to generate run-time code to clean the short integers so they comply with the Calling Standard.

[NO]EXACT_CDD_OFFSETS

The default is /ASSUME=NOEXACT_CDD_OFFSETS.

If /ASSUME=EXACT_CDD_OFFSETS is specified, the records input from the CDD are given the exact alignment (relative to the start of the record) specified by the CDD definition. This alignment is independent of the current compiler member-alignment setting.

If /ASSUME=NOEXACT_CDD_OFFSETS is specified, the compiler may modify the offsets specified in a CDD record according to the current member-alignment setting.

[NO]HEADER_TYPE_DEFAULT

The default is /ASSUME=HEADER_TYPE_DEFAULT.

In past versions of the C compiler, the `#include` directive always supplied a default file type of .h for C compilations. Similarly, the C++ compiler supplied a default file type of .hxx for C++ compilations.

However, the C++ standard requires that, for example, `#include <iostream>` be distinguishable from `#include <iostream.hxx>`. This is not possible with the header file-type default mechanism in effect.

You can disable the type default mechanism for either VSI C or VSI C++ by specifying /ASSUME=NOHEADER_TYPE_DEFAULT.

With /ASSUME=NOHEADER_TYPE_DEFAULT specified, an `#include` directive written with the standard syntax for header name (enclosed in quotes or angle brackets) will use the filename as specified, without supplying a default file type. More precisely stated, the default file type will be empty (just ".").

For example, a directory might contain three files named IOSTREAM., IOSTREAM.HXX, and IOSTREAM.H. By default, the C++ compiler processes `#include <iostream>` such that the file IOSTREAM.HXX is found, while the C compiler would find IOSTREAM.H.

However, if /ASSUME=NOHEADER_TYPE_DEFAULT is specified, the same directive causes the file IOSTREAM. to be found by both compilers, and the only way to include the file named IOSTREAM.HXX or IOSTREAM.H is to specify the .hxx or .h file type explicitly in the `#include` directive. Be aware that while the OpenVMS operating system treats filenames as case-insensitive and

normally displays them in uppercase, filenames in `#include` directives should use lowercase for best portability. This is more in keeping with other C and C++ implementations.

[NO]MATH_ERRNO

The default is /ASSUME=MATH_ERRNO, which does not allow intrinsic code for such math functions to be generated, even if /OPTIMIZE=INTRINSICS is in effect. Their prototypes and call formats, however, are still checked.

[NO]POINTERS_TO_GLOBALS

The default is /ASSUME=POINTER_TO_GLOBALS, which directs the compiler to assume that global variables have had their addresses taken in separately compiled modules and that, in general, any pointer dereference could be accessing the same memory as any global variable. This is often a significant barrier to optimization.

The /ANSI_ALIAS command-line qualifier allows some resolution based on data type, but /ASSUME=NOPOINTER_TO_GLOBALS provides significant additional resolution and improved optimization in many cases.

/ASSUME=NOPOINTER_TO_GLOBALS tells the compiler that any global variable accessed through a pointer in the compilation must have had its address taken within that compilation. The compiler can see any code that takes the address of an extern variable. If it does not see the address of the variable being taken, the compiler can assume that no pointer points to the variable.

Consider the following code sequence:

```
extern int x;
...
int *p;
...
*p = 3;
```

Under /ASSUME=NOPOINTERS_TO_GLOBALS, the compiler can assume that `x` is not changed by the assignment through `p` when generating code. This can lead to faster code.

In combination with the /PLUS_LIST_OPTIMIZE qualifier, several source modules can be treated as a single compilation for the purpose of this analysis. Because run-time libraries such as the VSI C RTL do not take the addresses of global variables defined in user programs, source modules can often be combined into a single compilation that allows /ASSUME=NOPOINTER_TO_GLOBALS to be used effectively.

Be aware that /ASSUME=NOPOINTERS_TO_GLOBALS does *not* tell the compiler that the compilation never uses pointers to access global variables (which is seldom true of real C programs).

[NO]WEAK_VOLATILE

This option affects the generation of code for assignments to objects that are less than or equal to 16 bits in size (for example: char, short) that have been declared as volatile.

Specifying /ASSUME=WEAK_VOLATILE directs the compiler to generate code for volatile assignments to single bytes or words without using the load-locked store-conditional sequences that, in general, are required to assure volatile data integrity when direct byte or word memory-access instructions are not being used.

This option is intended for use in special I/O hardware access situations, and should not generally be used.

The default is /ASSUME=NOWEAK_VOLATILE, which uses interlocked instructions for sub-longword volatile accesses when byte or word instructions are not enabled.

[NO]WHOLE_PROGRAM

The default is /ASSUME=NOWHOLE_PROGRAM.

The optimizations enabled by /ASSUME=WHOLE_PROGRAM include all those enabled by /ASSUME=NOPOINTER_TO_GLOBALS, and possibly additional optimizations as well.

[NO]WRITABLE_STRING_LITERALS

For /STANDARD=VAXC or /STANDARD=COMMON, the default is /ASSUME=WRITABLE_STRING_LITERALS.

For all other compiler modes, the default is /ASSUME=NOWRITABLE_STRING_LITERALS.

/[NO]CHECK[= ([NO]UNINITIALIZED_VARIABLES, [NO]BOUNDS [NO]POINTER_SIZE[=(option,...)]), [NO]FP_MODE(I64 only)

This qualifier is for use as a debugging aid.

/CHECK=UNINITIALIZED_VARIABLES

/CHECK=UNINITIALIZED_VARIABLES initializes all automatic variables to the value 0xfffa5a5afffa5a5a. This value is a floating NaN and, if used, causes a floating-point trap. If used as a pointer, this value is likely to cause an ACCVIO.

/CHECK=BOUNDS

/CHECK=BOUNDS enables run-time checking of array bounds. Array-bounds processing is performed in the following way:

- Checks are done only when accessing an array.
- Checks are not done when accessing a pointer, even if that access is done using the subscript operator. This means that checks are not done on arrays declared as formal parameters because they are considered pointers in the C language. If a formal parameter is a multi-dimension array, all bounds except the first are checked.
- If an array is accessed using the subscript operator (as either the left or right operand), and the subscript operator *is not* the operand of an address-of operator, the check is for the index to be between 0 and the number of array elements minus one, inclusive.
- If an array is accessed using the subscript operator (as either the left or right operand), and the subscript operator *is* the operand of the address-of operator, the check is for the index to be between 0 and the number of elements in the array, inclusive.

  The reason for treating the address-of case differently is that it is common programming practice to have a loop such as:

  ```
  int a[10];
  int *b;
  for (b = a ; b < &a[10] ; b++) { .... }
  ```

  In this case, access to &a[10] is allowed even though it is outside the range of the array.

- If the array is being accessed using pointer addition, the check is for the value being added to be between 0 and the number of elements in the array, inclusive.
- If the array is being accessed using pointer subtraction (that is, the subtraction of an integer value from a pointer, not the subtraction of one pointer from another), the check is for the value being subtracted to be between the negation of the number of elements in the array and 0, inclusive.
- In the previous three cases, an optional compile-time message (ident SUBSCRBOUNDS2) can be enabled to detect the case where an array has been accessed using either a constant subscript or constant pointer arithmetic, and the element accessed is exactly one past the end of the array.
- Bounds checking is not done for arrays declared with one element. (Because standard C does not allow arrays without dimensions inside `structs`, it is common practice to declare such arrays with a bounds specifier of 1.)

  In this case, an optional compile-time message (ident SUBSCRBOUNDS1) can be enabled to detect the case where an array declared with a single element is accessed using either a constant subscript or constant pointer arithmetic, and the element accessed is not part of the array.
- VSI C emits run-time checks for arrays indexed by constants, even though the compiler can and does detect this situation at compile-time. An exception is that no run-time check is made if the compiler can determine that the access is valid.
- Here are examples of some array references:

```
int a[10];
int *b;
int c;

int *d;
int vla[c];
int one[1];

a[c] = 1;           // check c is from 0-9
b[c] = 1;           // no check
c[a] = 1;           // check c is from 0-9
b = &a[c]           // check c is from 0-10
*(a + c) = 1;       // check c is from 0-10
*(a - c) = 1;       // check c is from -10 to 0
d = a + c;          // check that c is from 0-10
d = b + c;          // no check
a[1] = 1;           // no run-time check - know access is valid
vla[1] = 1;         // run-time check
a[10] = 1;          // run-time check  (and compiler diagnostic)
d = a + 10;         // no run-time check, optional SUBSCRBOUNDS2
                    //    message can be enabled
c = one[5];         // no run-time check, optional SUBSCRBOUNDS1
                    //    message can be enabled
```

- If a multi-dimension array is accessed, the compiler performs checks on each of the subscript expressions, making sure each is within the corresponding bound. So for the following code, the compiler checks that both `x` and `y` are between 0 and 9. It does not check that `10 * x + y` is between 0 and 99:

```
int a[10][10];
int x,y,z;
```

```
x = a[x][y];
```

Notes

- Because of operating system differences, the behavior of the run-time array-bounds checking is different on *UNIX* systems than on OpenVMS systems.

  If there is no handler, an OpenVMS program fails with:

  ```
  %SYSTEM-F-SUBRNG, arithmetic trap, subscript out of range at
   PC=xxx, PS=xxx
  %TRACE-F-TRACEBACK, symbolic stack dump follows
  ```

  On *UNIX* systems, the output would be:

  ```
  Trace/BPT trap (core dumped)
  ```

  Furthermore, to trap the error on OpenVMS systems, a user needs to write:

  ```
  signal(SIGFPE, handler);
  ```

  While on *UNIX* systems, the equivalent line would be:

  ```
  signal(SIGTRAP, handler);
  ```

- When run-time checking is enabled, the VSI C compiler emits a bad check in certain cases. These cases arise when an array is accessed using pointer arithmetic and run-time array-bounds checking is enabled. In such a case, the compiler can output only the checking code for the first pointer-arithmetic operation performed on the array. This can result in an incorrect check if the resulting pointer value is again operated on by pointer arithmetic.

  Consider the following expression where `a` is a pointer, `c` is an array, and `c` and `d` are integers:

  ```
  a = b + c - d;
  ```

  When bounds checking is enabled, the compiler outputs a check to verify that `c` is within the bounds of the array. This leads to an incorrect run-time trap in cases where `c` is outside the bounds of the array and `c - d` is not.

  In these cases, the compiler outputs a diagnostic noting that the check code it produced is bad. You can then recode the pointer expression so that the integer part is in parentheses. In this way, the expression will contain only one pointer-arithmetic operation, and the compiler will output the correct check. In the previous example, the expression would be changed to:

  ```
  a = b + (c - d);
  ```

/CHECK=POINTER_SIZE

/CHECK=POINTER_SIZE directs the compiler to generate code that checks 64-bit pointer values (used in certain contexts where 32-bit pointers are also present) to make sure they will fit in a 32-bit pointer. If such a value cannot be represented by a 32-bit pointer, the run-time code signals a range error (SS$_RANGEERR).

To control the types of pointer-size checks you want made, use one or more of the POINTER_SIZE option keywords shown in *Table 1.5, "/CHECK=POINTER_SIZE Qualifier Options"*.

Table 1.5. /CHECK=POINTER_SIZE Qualifier Options

Option	Usage
[NO]ASSIGNMENT	Check whenever a 64-bit pointer is assigned to a 32-bit pointer (including use as an actual argument).
[NO]CAST	Check whenever a 64-bit pointer is cast to a 32-bit pointer.
[NO]INTEGER_CAST	Check whenever a long pointer is cast to a 32-bit integer.
[NO]PARAMETER	Check all formal parameters at function startup to make sure that all formal parameters declared to be 32-bit pointers are 32-bit values.
ALL	Do all checks.
NONE	Do no checks.

Specifying /CHECK=POINTER_SIZE defaults
to /CHECK=POINTER_SIZE=(ASSIGNMENT,PARAMETER).

For information about compiler features that affect pointer size, see the following:

- /POINTER_SIZE
- `#pragma pointer_size`
- `#pragma required_pointer_size`
- `__INITIAL_POINTER_SIZE` predefined macro

The following contrived program contains a number of pointer assignments. The comment on each line indicates what /CHECK=POINTER_SIZE keyword to specify to enable checking for that line.

```
#pragma required_pointer_size long
int *a;
char *b;
typedef char * l_char_ptr;

#pragma required_pointer_size short
char *c;
int *d;

foo(int * e)              /* Check e if PARAMETER is specified. */
{
    d = a;                /* Check a if ASSIGNMENT is specified. */
    c = (char *) a;       /* Check a if CAST is specified. */
    c = (char *) d;       /* No checking ever. */
    foo( a );             /* Check a if ASSIGNMENT is specified. */
    bar( a );             /* No checking ever - no prototype */
    b = (l_char_ptr) a;   /* No checking ever. */
    c = (l_char_ptr) a;   /* Check a if ASSIGNMENT is specified */
    b = (char *) a;       /* Check if CAST is specified. */
}
```

/CHECK=[NO]FP_MODE (I64 only)

/CHECK=FP_MODE generates code in the prologue of every function defined in the compilation to compare the current values of certain fields in the processor's floating-point status register (FPSR) with the values expected in those fields based on the command-line qualifiers with which the function was compiled.

The values checked are the rounding mode and the trap-enable bits:

- If the rounding mode is not consistent with the value of the /ROUNDING_MODE qualifier specified at compile time, an informational message SYSTEM-I-FPMODERC is issued at runtime, citing the current mode and the compile-time specified mode (Note that /ROUNDING_MODE=DYNAMIC is treated the same as /ROUNDING_MODE=NEAREST for this purpose).
- If the trap-enable flags are not consistent with the setting of the /IEEE qualifier (for /FLOAT=IEEE_FLOAT compilations) or with the setting used to implement VAX floating types (for /FLOAT=G_FLOAT or /FLOAT=D_FLOAT compilations), an informational message SYSTEM-I-FPMODECTL is issued at run time, citing the current trap-enable flags as well as the trap-enable flags expected by the compilation. To identify the point of failure, you need to rerun the program under DEBUG and issue "SET BREAK/EXCEPTION".

Note that the checking code generated for /CHECK=FP_MODE includes a standard call to OTS$CHECK_FP_MODE within the prologue of each function, and OTS$CHECK_FP_MODE itself assumes the standard calling conventions (described in the OpenVMS Calling Standard). Because of this, it is not possible to use this checking option when compiling function definitions that have a nonstandard linkage (see `#pragma linkage` and `#pragma use_linkage`) specifying conventional scratch registers with the PRESERVED or NOTUSED attribute. Doing so will cause the compiler to issue the "REGCONFLICT" E-level diagnostic at the opening brace of such function definitions. To compile such functions successfully, the FP_MODE keyword must be removed from the list of /CHECK= keywords.

Defaults

Omitting this qualifier defaults to /NOCHECK, which equates to /CHECK=(NOUNINITIALIZED_VARIABLE,NOBOUNDS,NOPOINTER_SIZE,NOFP_MODE).

Specifying /CHECK defaults to /CHECK=(UNINITIALIZED_VARIABLES,BOUNDS, POINTER_SIZE=(ASSIGNMENT,PARAMETER),FP_MODE).

`/[NO]COMMENTS=option`

Governs whether or not comments appear in preprocess output files and, if they are to appear, whether they appear themselves or are replaced by a single space.

Table 1.6, "/COMMENTS Qualifier Options" shows the /COMMENTS qualifier options.

Table 1.6. /COMMENTS Qualifier Options

Option	Usage
AS_IS	Specifies that the comment appears in the output file.
SPACE	Specifies that a single space replaces the comment in the output file.

/NOCOMMENTS specifies that nothing replaces the comment in the output file. This can result in inadvertent token pasting.

The VSI C preprocessor might replace a comment at the end of a line or on a line by itself with nothing, even if /COMMENTS=SPACE is specified. Doing so does not change the meaning of the program.

The default is /COMMENTS=SPACE for the ANSI89, RELAXED, and MIA modes of the compiler. The default is /NOCOMMENTS for all other compiler modes.

Specifying /COMMENTS on the command line defaults to /COMMENTS=AS_IS.

/[NO]CROSS_REFERENCE

Specifies whether the compiler generates cross-references for variable names.

If you specify /CROSS_REFERENCE, the compiler lists, for each variable referenced in the procedure, the line numbers of the lines on which the variable is referenced.

This qualifier has no effect unless you also specify /LIST and either /SHOW=SYMBOLS or /SHOW=BRIEF. The default is /NOCROSS_REFERENCE.

/[NO]DEBUG[=(option[,…])]

Includes information in the object module for use by the OpenVMS Debugger.

If the /DEBUG qualifier is not specified, the default is:

- /DEBUG=(TRACEBACK,NOSYMBOLS) on Alpha systems.
- /DEBUG=(TRACEBACK,NOINLINE,NOSYMBOLS) on VAX systems.

Specifying /DEBUG with no keywords is equivalent to specifying /DEBUG=ALL.

Table 1.7, "Debugger Compilation Options" describes the debugger options.

Table 1.7. Debugger Compilation Options

Option	Usage
ALL	Includes symbol table records and traceback records for both VAX and Alpha systems. On VAX systems, this also selects the behavior of the INLINE keyword. On Alpha and IA-64 systems, /DEBUG=ALL is equivalent to /DEBUG=(TRACEBACK,SYMBOLS). On VAX systems, /DEBUG=ALL is equivalent to /DEBUG=(TRACEBACK,SYMBOLS,INLINE).
INLINE (VAX only)	Generates debug information to cause a STEP command to STEP/INTO an inlined function call.
NOINLINE (VAX only)	Generates debug information to cause a STEP command to STEP/OVER an inlined function call.
NONE	Does not include any debugging information. This is equivalent to /NODEBUG.
NOTRACEBACK	Suppresses generation of traceback records.
NOSYMBOLS	Suppresses generation of symbol table records.
SYMBOLS	Generates symbol table records.
TRACEBACK	Generates traceback records.

/DECC

Invokes the VSI C compiler.

On OpenVMS VAX systems, the CC command is used to invoke either the VAX C or VSI C compiler. If your system has a VAX C compiler already installed on it, the VSI C installation procedure provides

the option of specifying which compiler will be invoked by default when just the CC command is used. To invoke the compiler that is not the default, use the CC command with the appropriate qualifier: CC/DECC for the VSI C compiler, or CC/VAXC for the VAX C compiler. If your system does not have a VAX C compiler installed on it, the CC command will invoke the VSI C compiler.

On OpenVMS Alpha and IA-64 systems, specifying /DECC is equivalent to not specifying it; this qualifier is supported to provide compatibility with VSI C on OpenVMS VAX systems.

/[NO]DEFINE= (identifier[=definition][,…])

/[NO]UNDEFINE= (identifier[,…])

Performs the same functions as the `#define` and `#undef` preprocessor directives. The /DEFINE qualifier defines a macro to be substituted for every occurrence of a given identifier in the compilation unit or units. The /UNDEFINE qualifier cancels a previous definition (but not subsequent ones). When both /DEFINE and /UNDEFINE are present in a compilation unit or on the CC command line, /DEFINE is evaluated before /UNDEFINE.

Since /DEFINE and /UNDEFINE are not part of the source file, they are not associated with a listing line number or source line number. Therefore, when an error occurs in a command-line definition, the message displayed at the terminal does not indicate a line number. In the listing file, these diagnostic messages are placed before the source listing in the order that they were encountered. When the expansion of a definition causes an error at a specific source line in the program, the diagnostics—both at the terminal and in the listing file—are associated with that source line.

A command line containing the /DEFINE and the /UNDEFINE qualifiers can be long. Continuation characters cannot appear within quotes or they will be included in the macro stream. The length of a CC command line cannot exceed the maximum length allowed by DCL.

The /NODEFINE and /NOUNDEFINE qualifiers are provided for compatibility with other DCL qualifiers. You can use these qualifiers to cancel /DEFINE or /UNDEFINE qualifiers that you have specified in a symbol that you use to compile VSI C programs.

The defaults are /NODEFINE and /NOUNDEFINE.

Usage and Examples

Since the CC command line must be compatible with DCL, the syntax of the /DEFINE and /UNDEFINE qualifiers differs from the syntax of the `#define` and `#undef` preprocessor directives in the following way:

- An equal sign is required after /DEFINE; a space is required after `#define`. For example, the following are equivalent:

  ```
  $ CC/DEFINE=TRUE
  #define TRUE 1
  ```

  Note that the value of `TRUE` on the /DEFINE qualifier is automatically set to 1. Any other value must be specified. For example, the following are equivalent:

  ```
  $ CC/DEFINE=MAYBE=2
  #define MAYBE 2
  ```

- DCL converts all input to uppercase unless it is enclosed in quotation marks. For example, the following are equivalent:

  ```
  $ CC/DEFINE=true
  #define TRUE 1
  ```

- The macro defined on the /DEFINE qualifier must be enclosed in quotation marks if at least one of the following is true:
  - You want to preserve lowercase
  - The macro definition contains spaces or characters that would not be valid on the DCL command line.
  - The macro is a function-like macro

  For example:

```
$ CC/DEFINE="true"         ! Preserves lowercase
$ CC/DEFINE="blank=' '"    ! Contains and preserves the blank
$ CC/DEFINE="f1=a+b"       ! Contains a '+' character
$ CC/DEFINE="funct(a)=2"   ! Defines a function-like macro
```

- Within a macro definition and inside quotation marks, a delimiter can be either an equal sign or a space, whichever comes first. If an equal sign is the delimiter, the following examples are equivalent:

```
$ CC/DEFINE="true=1"
#define true 1
```

  If a space is the delimiter, the following examples are equivalent:

```
$ CC/DEFINE="true =1"
#define true =1
```

  In this example, the space, preserved by the quotation marks, serves as the delimiter, assigning `true` a value of `=1`, which is clearly not intended.

- Within a definition and outside quotation marks, the only allowed delimiter is an equal sign; a space terminates the definition. The following definitions, for example, are not recognized by DCL:

```
$ CC/DEFINE= TRUE
$ CC/DEFINE=(FALSE 0
```

  In the first example, DCL interprets `TRUE` as a file specification; in the second, DCL flags an invalid value specification.

- When more than one /DEFINE is present on the CC command line or in a single compilation unit, only the last /DEFINE is used. Similarly, only the last /UNDEFINE on the CC command line or the compilation unit is used.

You can pass an equal sign to the compiler in any of the following ways:

```
$ CC/DEFINE=(EQU==,"equ =","equal==")
```

In the first definition, the first equal sign is removed by DCL as the delimiter; the second equal sign is passed to the compiler. In the second example, the space is recognized as a delimiter because the definition is inside quotes; therefore, only one equal sign is required. In the third definition, the first equal sign is recognized as the delimiter and is removed; the second equal sign is passed to the compiler.

You can pass quotation marks in any of the following ways:

```
$ CC/DEFINE=(QUOTES="""","funct(b)=printf(")")
```

In both examples, DCL removes the first and last quotation marks before passing the definition to the compiler.

Here is a simple use of the /UNDEFINE qualifier to cancel a previous definition of TRUE:

```
$ CC/UNDEFINE=TRUE
```

The /UNDEFINE qualifier is useful for undefining the predefined VSI C preprocessor constants. For example, if you use a preprocessor system identification macro (such as __vaxc, __VAXC, __DECC, or __vms) to conditionally compile segments of VSI C specific code, you can undefine that constant to see how the portable sections of your program execute. Consider the following program:

```
main()
{
#if __DECC
printf("I'm being compiled with VSI C on an OpenVMS system.");
#else
printf("I'm being compiled on some other compiler.");
#endif
}
```

This program produces the following output:

```
$ CC  EXAMPLE.C
$ LINK  EXAMPLE.OBJ
$ RUN  EXAMPLE.EXE
I'm being compiled with VSI C on an OpenVMS system.

$ CC/UNDEFINE="DECC" EXAMPLE

$ LINK  EXAMPLE.OBJ

$ RUN  EXAMPLE.EXE
I'm being compiled on some other compiler.
```

/[NO]DIAGNOSTICS[=file-spec]

Creates a file containing compiler messages and diagnostic information. The default file extension for a diagnostics file is .DIA. The diagnostics file is used with the Language-Sensitive Editor (LSE). To display a diagnostics file, enter the command REVIEW/FILE=file-spec while in LSE. For more information, see *Appendix C, "Programming Tools"*. The default is /NODIAGNOSTICS.

/ENDIAN=option

This qualifier takes the options BIG or LITTLE.

It controls whether big or little endian ordering of bytes is carried out in character constants. For example, consider the following declaration:

```
int foo = 'ABCD';
```

Specifying /ENDIAN=LITTLE places 'A' in the first byte, 'B' in the second byte, and so on.

Specifying /ENDIAN=BIG places 'D' in the first byte, 'C' in the second byte, and so on.

The default is /ENDIAN=LITTLE.

/[NO]ERROR_LIMIT[=n]

This qualifier limits the number of Error-level diagnostic messages that are acceptable during program compilation. Compilation terminates when the limit *n* is exceeded. /NOERROR_LIMIT specifies that there is no limit on error messages.

The default is /ERROR_LIMIT=30, which specifies that compilation terminates after 31 error messages.

/EXTERN_MODEL=option

In conjunction with the /[NO]SHARE_GLOBALS qualifier, controls the initial compiler model for external objects. Conceptually, the compiler behaves as if the first line of the program being compiled was a `#pragma extern_model` with the model and psect name, if any, specified by the /EXTERN_MODEL qualifier and with the `shr` or `noshr` keyword specified by the /[NO]SHARE_GLOBALS qualifier.

For example, assume the command line contains the following qualifiers:

```
/EXTERN_MODEL=STRICT_REFDEF="MYDATA"/NOSHARE
```

The compiler will behave as if the program begins with the following line:

```
#pragma extern_model strict_refdef "MYDATA" noshr
```

Table 1.8, "/EXTERN_MODEL Qualifier Options" describes the /EXTERN_MODEL qualifier options.

Table 1.8. /EXTERN_MODEL Qualifier Options

Option	Usage
COMMON_BLOCK	Sets the compiler's `extern_model` to the `common_block` model. This is the model traditionally used for `extern` data by VAX C.
RELAXED_REFDEF	Sets the compiler's `extern_model` to the `relaxed_refdef` model. Some declarations are references and some are definitions. Multiple uninitialized definitions for the same object are allowed and are resolved into one by the linker. However, a reference requires that at least one definition exist. This is the model used by the portable C compiler (`pcc`) on UNIX systems.
STRICT_REFDEF [="*name*"]	Sets the compiler's `extern_model` to the `strict_refdef` model. Some declarations are references and some are definitions. There must be exactly one definition in the program for any symbol referenced. The optional *name*, in quotation marks, is the name of the psect for any definitions. This is the model specified by standard C. Use it in a program that is to be a strict standard-conforming program. This model is the preferred alternative to the nonstandard storage-class keywords `globaldef` and `globalref`.
GLOBALVALUE	Sets the compiler's `extern_model` to the `globalvalue` model. This model is similar to the `strict_refdef` model except that these global objects have no storage; instead, they are link-time constant values. There are two cases:

Option	Usage
	• If the declaration is a standard C reference, the same object file records are produced as VAX C would produce for an uninitialized `globalvalue`. • If the declaration is a standard C definition, the same object records are produced as VAX C would produce for an initialized `globalvalue`. This model is the preferred alternative to the nonstandard storage-class keyword `globalvalue`.

The default is /EXTERN_MODEL=RELAXED_REFDEF. This is different from VAX C, which uses the common block model for external objects.

/[NO]FIRST_INCLUDE=(file[,…])

Includes the specified files before any source files. This qualifier corresponds to the *UNIX* `-FI` switch.

This qualifier is useful if you have command lines to pass to the C compiler that are exceeding the DCL command-line length limit. Using the /FIRST_INCLUDE qualifier can help solve this problem by replacing lengthy /DEFINE and /WARNINGS qualifiers with `#define` and `#pragma message` preprocessor directives placed in a /FIRST_INCLUDE file.

When /FIRST_INCLUDE=*file* is specified, *file* is included in the source as if the line before the first line of the source was:

`#include "`*file*`"`

If more than one file is specified, the files are included in their order of appearance on the command line.

The default is /NOFIRST_INCLUDE.

/FLOAT=option

Controls the format of floating-point variables.

Table 1.9, "/FLOAT Qualifier Options" describes the /FLOAT qualifier options.

Table 1.9. /FLOAT Qualifier Options

Option	Usage
D_FLOAT	`double` variables are represented in D_floating format. The __D_FLOAT macro is predefined.
G_FLOAT	`double` variables are represented in G_floating format. The __G_FLOAT macro is predefined.
IEEE_FLOAT	`float` and `double` variables are represented in IEEE floating-point format (S_float and T_float, respectively). The __IEEE_FLOAT macro is predefined. Use the /IEEE_MODE qualifier for controlling the handling of IEEE exceptional values. If /IEEE_MODE is not specified, the default behavior is /IEEE_MODE=FAST for Alpha systems and /IEEE_MODE=DENORM_RESULTS for IA-64 systems.

OpenVMS VAX Systems (VAX only)

On OpenVMS VAX systems, representation of `double` variables defaults toD_floating format if not overridden by another format specified with the /FLOAT or /[NO]G_FLOAT qualifier. There is one exception: if /STANDARD=MIA is specified, G_floating is the default. If you are linking against object-module libraries, a program compiled with G_floating format must be linked with the object library DECCRTLG.OLB. (VAX only)

OpenVMS Alpha Systems (Alpha only)

On OpenVMS Alpha systems, representation of `double` variables defaults to G_floating format if not overridden by another format specified with the /FLOAT or /[NO]G_FLOAT qualifier.

If you are linking against object-module libraries, and /PREFIX=ALL is not specified on the command line, then a program compiled with:

- G_FLOAT format must be linked with the object library VAXCRTL.OLB
- D_FLOAT format must be linked with VAXCRTLD.OLB
- IEEE_FLOAT format must be linked with VAXCRTLT.OLB

The VAXCRTLX.OLB, VAXCRTLDX.OLB, and VAXCRTLTX.OLB libraries are used for the same floating-point formats, respectively, but include support for X_FLOAT format (/L_DOUBLE_SIZE=128).

If /PREFIX=ALL is specified, then there is no need to link to the above-mentioned *.OLB object libraries. All the symbols you need are in STARLET.OLB.

IA-64 systems (I64 only)

This section describes floating-point support and application porting considerations for IA-64 systems.

On OpenVMS IA-64 systems, /FLOAT=IEEE_FLOAT is the default floating-point representation. IEEE format data is assumed and IEEE floating-point instructions are used. There is no hardware support for floating-point representations other than IEEE, although you can specify the /FLOAT=D_FLOAT or /FLOAT=G_FLOAT compiler option. These VAX floating-point formats are supported in the I64 compiler by generating run-time code that converts VAX floating-point formats to IEEE format to perform arithmetic operations, and then converts the IEEE result back to the appropriate VAX floating-point format. This imposes additional run-time overhead and some loss of accuracy compared to performing the operations in hardware on Alpha and VAX systems. The software support for the VAX formats is provided to meet an important functional compatibility requirement for certain applications that need to deal with on-disk binary floating-point data.

on IA-64 systems, the default for /IEEE_MODE is DENORM_RESULTS, which is a change from the default of /IEEE_MODE=FAST on Alpha systems. This means that by default, floating-point operations may silently generate values that print as Infinity or Nan (the industry-standard behavior), instead of issuing a fatal run-time error as they would when using VAX floating-point format or /IEEE_MODE=FAST. Also, the smallest-magnitude nonzero value in this mode is much smaller because results are allowed to enter the denormal range instead of being flushed to zero as soon as the value is too small to represent with normalization.

The conversion of VAX floating-point formats to IEEE single and IEEE double floating-point types on the Intel Itanium architecture is a transparent process that will not impact most applications. All you need to do is recompile your application. Because IEEE floating-point format is the default, unless your build explicitly specifies VAX floating-point format options, a simple rebuild for IA-64 systems will use

the native IEEE formats directly. For the large class of programs that do not directly depend on the VAX formats for correct operation, this is the most desirable way to build for IA-64 systems.

When you compile an OpenVMS application that specifies an option to use VAX floating-point on an IA-64 system, the compiler automatically generates code for converting floating-point formats. Whenever the application performs a sequence of arithmetic operations, this code does the following:

1. Converts VAX floating-point formats to either IEEE single or IEEE double floating-point formats.
2. Performs arithmetic operations in IEEE floating-point arithmetic.
3. Converts the resulting data from IEEE formats back to VAX formats.

Where no arithmetic operations are performed (VAX float fetches followed by stores), no conversion will occur. The code handles such situations as moves.

VAX floating-point formats have the same number of bits and precision as their equivalent IEEE floating-point formats. For most applications the conversion process will be transparent and thus a non-issue.

In a few cases, arithmetic calculations might have different results because of the following differences between VAX and IEEE formats:

- Values of numbers represented
- Rounding rules
- Exception behavior

These differences might cause problems for applications that do any of the following:

- Depend on exception behavior
- Measure the limits of floating-point behaviors
- Implement algorithms at maximal processor-specific accuracy
- Perform low-level emulations of other floating-point processors
- Use direct equality comparisons between floating-point values, instead of appropriately ranged comparisons (a practice that is extremely vulnerable to changes in compiler version or compiler options, as well as architecture)

You can test an application's behavior with IEEE floating-point values by compiling it on an OpenVMS Alpha system using /FLOAT=IEEE_FLOAT/IEEE_MODE=DENORM. If that produces acceptable results, then simply build the application on the OpenVMS IA-64 system using the same qualifier.

If you determine that simply recompiling with an /IEEE_MODE qualifier is not sufficient because your application depends on the binary representation of floating-point values, then first try building for your IA-64 system by specifying the VAX floating-point option that was in effect for your VAX or Alpha build. This causes the representation seen by your code and on disk to remain unchanged, with some additional run-time cost for the conversions generated by the compiler. If this is not an efficient approach for your application, you can convert VAX floating-point binary data in disk files to IEEE floating-point formats before moving the application to an IA-64 system.

`/GRANULARITY=option`

Controls the size of shared data in memory that can be safely accessed from different threads. The possible size values are BYTE, LONGWORD, and QUADWORD.

Specifying BYTE allows single bytes to be accessed from different threads sharing data in memory without corrupting surrounding bytes. This option will slow run-time performance.

Specifying LONGWORD allows naturally aligned 4-byte longwords to be accessed safely from different threads sharing data in memory. Accessing data items of 3 bytes or less, or unaligned data, may result in data items written from multiple threads being inconsistently updated.

Specifying QUADWORD allows naturally aligned 8-byte quadwords to be accessed safely from different threads sharing data in memory. Accessing data items of 7 bytes or less, or unaligned data, might result in data items written from multiple threads being inconsistently updated. This is the default.

/IEEE_MODE=option

Selects the IEEE floating-point mode to be used if /FLOAT=IEEE_FLOAT is specified.

Table 1.10, "/IEEE_MODE Options" describes the /IEEE_MODE options.

Table 1.10. /IEEE_MODE Options

Option	Usage
FAST	During program execution, only finite values (no infinities, NaNs, or denorms) are created. Underflows and denormal values are flushed to zero. Exceptional conditions, such as floating-point overflow, divide-by-zero, or use of an IEEE exceptional operand are fatal.
UNDERFLOW_TO_ZERO	Generate infinities and NaNs. Flush denormalized results and underflow to zero without exceptions.
DENORM_RESULTS	Same as UNDERFLOW_TO_ZERO, except that denorms are generated.
INEXACT	Same as DENORM_RESULTS, except that inexact values are trapped. This is the slowest mode, and is not appropriate for any sort of general-purpose computations.

On Alpha systems, the default is /IEEE_MODE=FAST.

on IA-64 systems, the default is /IEEE_MODE=DENORM_RESULTS.

The INFINITY and NAN macros defined in `<math.h>` are available to programs compiled with /FLOAT=IEEE and /IEEE_MODE={*anything other than FAST*}, and in a compiler mode that enables C99 extensions in the headers (any mode other than COMMON or VAXC).

On Alpha systems, the /IEEE_MODE qualifier generally has its greatest effect on the generated code of a compilation. When calls are made between functions compiled with different /IEEE_MODE qualifiers, each function produces the /IEEE_MODE behavior with which it was compiled.

on IA-64 systems, the /IEEE_MODE qualifier primarily affects only the setting of a hardware register at program startup. In general, the /IEEE_MODE behavior for a given function is controlled by the /IEEE_MODE option specified on the compilation that produced the main program: the startup code for the main program sets the hardware register according the command-line qualifiers used to compile the main program.

When applied to a compilation that does not contain a main program, the /IEEE_MODE qualifier does have some effect: it might affect the evaluation of floating-point constant expressions, and it is used to set the EXCEPTION_MODE used by the math library for calls from that compilation. But the qualifier

has no effect on the exceptional behavior of floating-point calculations generated as inline code for that compilation. Therefore, if floating-point exceptional behavior is important to an application, all of its compilations, including the one containing the main program, should be compiled with the same /IEEE_MODE setting.

Even on Alpha systems, the particular setting of /IEEE_MODE=UNDERFLOW_TO_ZERO has this characteristic: its primary effect requires the setting of a run-time status register, and so it needs to be specified on the compilation containing the main program in order to be effective in other compilations.

/[NO]INCLUDE_DIRECTORY= (pathname[,…])

Provides similar functionality to the `-I` option of the cc command on *UNIX* systems. This qualifier allows you to specify additional places to search for include files. A place can be one of the following:

- OpenVMS file-spec to be used as a default file-spec to RMS file services (example: DISK$:[directory])
- UNIX style pathname in quotation marks (example: "/sys")
- Empty string ("")

If one of the places is specified as an empty string, the compiler does not search any of its conventionally-named places:

DECC$USER_INCLUDE
DECC$SYSTEM_INCLUDE
DECC$LIBRARY_INCLUDE
SYS$COMMON:[DECC$LIB.INCLUDE.*]
DECC$TEXT_LIBRARY
SYS$LIBRARY:DECC$RTLDEF.TLB
SYS$LIBRARY:SYS$STARLET_C.TLB

Instead, it searches only places specified explicitly on the command line by the /INCLUDE_DIRECTORY and /LIBRARY qualifiers (or by the location of the primary source file, depending on the /NESTED_INCLUDE_DIRECTORY qualifier). This behavior is similar to that obtained by specifying -I without a directory name to the *UNIX* `cc` command.

The basic search order depends on the form of the header-file name (after macro expansion). Additional aspects of the search order are controlled by other command-line qualifiers and the presence or absence of logical name definitions.

Only the portable forms of the `#include` directive are affected by the pathnames specified on an /INCLUDE_DIRECTORY qualifier:

- In quotes (example: `#include "stdio.h"`)
- In angle brackets (example: `#include <stdio.h>`)

However, an empty string also affects the text-module form specific to OpenVMS systems (example: `#include stdio`).

Except where otherwise specified, searching a "place" means that the string designating the place is used as the default file-spec in a call to an RMS system service (for example, $SEARCH/$PARSE). The file-spec consists of the name in the `#include` directive without enclosing delimiters. The search terminates successfully as soon as a file can be opened for reading.

Note

Prior to OpenVMS VAX Version 7.1, the operating system did not provide a SYS$LIBRARY:SYS$STARLET_C.TLB nor the headers contained therein. Instead, the compiler installation generated these headers and placed them in SYS$LIBRARY:DECC$RTLDEF.TLB.

Quoted Form

For the quoted form of inclusion, the search order is:

1. One of the following:

   - If /NESTED_INCLUDE_DIRECTORY=INCLUDE_FILE (the default) is in effect, search the directory containing the file in which the `#include` directive itself occurred. The *directory containing* means the RMS resultant string obtained when the file in which the `#include` occurred was opened, except that the filename and subsequent components are replaced by the default file type for headers (".h", or just "." if /ASSUME=NOHEADER_TYPE_DEFAULT is in effect). The resultant string will not have translated any concealed device logical.

   - If /NESTED_INCLUDE_DIRECTORY=PRIMARY_FILE is in effect, search the default file type for headers using the context of the primary source file. This means that just the file type (".h" or ".") is used for the default file-spec but, in addition, the chain of "related file-specs" used to maintain the sticky defaults for processing the next top-level source file is applied when searching for the include file. This most closely matches the behavior of the VAX C compiler.

   - If /NESTED_INCLUDE_DIRECTORY=NONE is in effect, this entire step (Step 1) is bypassed.

2. Search the places specified in the /INCLUDE_DIRECTORY qualifier, if any. A place that can be parsed successfully as an OpenVMS file-spec and that does not contain an explicit file type or version specification is edited to append the default header file type specification (".h" or ".").

   A place containing a "/" character is considered to be a UNIX-style name. If the name in the `#include` directive also contains a "/" character that is not the first character and is not preceded by a "!" character (it is not an absolute UNIX-style pathname), then the name in the `#include` directive is appended to the named place, separated by a "/" character, before applying the decc$to_vms pathname translation function. The result of the decc$to_vms translation is then used as the filespec to try to open.

3. If DECC$USER_INCLUDE is defined as a logical name, search DECC$USER_INCLUDE:.H, or just DECC$USER_INCLUDE:. if /ASSUME=NOHEADER_TYPE_DEFAULT is in effect.

4. If the file is not found, follow the steps for the angle-bracketed form of inclusion.

Angle-Bracketed Form

For the angle-bracketed form of inclusion, the search order is:

1. Search the place "/". This is a UNIX-style name that can combine only with UNIX names specified explicitly in the `#include` directive. It causes a specification like to be considered first as /sys/types.h, which is translated by decc$to_vms to SYS:TYPES.H.

2. Search the places specified in the /INCLUDE_DIRECTORY qualifier, exactly as in Step 2 for the quoted form of inclusion.

3. If DECC$SYSTEM_INCLUDE is defined as a logical name, search DECC$SYSTEM_INCLUDE:.H, or just DECC$SYSTEM_INCLUDE:. if /ASSUME=NOHEADER_TYPE_DEFAULT is in effect.

4. If DECC$LIBRARY_INCLUDE is defined as a logical name and DECC$SYSTEM_INCLUDE is not defined as a logical name, search DECC$LIBRARY_INCLUDE:.H, or just DECC$LIBRARY_INCLUDE:. if /ASSUME=NOHEADER_TYPE_DEFAULT is in effect.

5. If neither DECC$LIBRARY_INCLUDE nor DECC$SYSTEM_INCLUDE are defined as logical names, then search the default list of places for plain text-file copies of compiler header files as follows:

   SYS$COMMON:[DECC$LIB.INCLUDE.DECC$RTLDEF]*.H
   SYS$COMMON:[DECC$LIB.INCLUDE.SYS$STARLET_C]*.H

   Note

   The compiler installation does not create these directories of header files. Instead, it creates [DECC$LIB.REFERENCE] for your convenience. But if you choose to create and populate SYS$COMMON:[DECC$LIB.INCLUDE.DECC$RTLDEF] or SYS$COMMON:[DECC$LIB.INCLUDE.SYS$STARLET_C], the compiler will search them.

   If the file is not found, perform the text library search described in the next step.

6. Extract the simple filename and file type from the `#include` specification and use the filename as the module name to search a list of text libraries associated with that file type.

   For any file type, the initial text libraries searched consist of those named on the command line with /LIBRARY qualifiers, searched in left-to-right order.

   If the /INCLUDE_DIRECTORY qualifier contained an empty string, no further text libraries are searched. Otherwise, DECC$TEXT_LIBRARY is searched for all file types.

   If DECC$LIBRARY_INCLUDE is defined as a logical name, then no further text libraries are searched. Otherwise, the subsequent libraries searched for each file type are:

   - For a file type of ".h" or ".":

     SYS$LIBRARY:DECC$RTLDEF.TLB
     SYS$LIBRARY:SYS$STARLET_C.TLB

   - For a file type other then ".h" or ".":

     SYS$LIBRARY:SYS$STARLET_C.TLB

7. If the previous step fails, search the following:

   SYS$LIBRARY:.H

   Under /ASSUME=NOHEADER_TYPE_DEFAULT, the default file type is modified as usual.

Text-Module Form

For the text-module (nonportable) form of inclusion, the name can only be an identifier. It, therefore, has no associated file type.

The identifier is used as a module name to search the following:

1. The text libraries named on the command line with /LIBRARY qualifiers, in left-to-right order.

2. The following list of text libraries in the order shown (unless the /INCLUDE_DIRECTORY qualifier contains an empty string, in which case no further text libraries are searched):

   DECC$TEXT_LIBRARY
   SYS$LIBRARY:DECC$RTLDEF.TLB
   SYS$LIBRARY:SYS$STARLET_C.TLB

The default for this qualifier is /NOINCLUDE_DIRECTORY.

/L_DOUBLE_SIZE=option

Determines how the compiler interprets the `long double` type. The qualifier options are 64 and 128.

Specifying /L_DOUBLE_SIZE=64 treats all `long double` references as G_FLOAT, D_FLOAT, or T_FLOAT, depending on the value of the /FLOAT qualifier.

Specifying /L_DOUBLE_SIZE=128 treats all `long double` references as X_FLOAT.

The default is /L_DOUBLE_SIZE=128.

/LIBRARY

Indicates that the associated input file is a library containing modules of VSI C source text. If the library specification does not include a file extension, the CC command line assumes the .TLB default type. You must join the /LIBRARY qualifier with a file specification in a compilation unit using a plus sign (+); you cannot place the qualifier at other places on the CC command line. No matter where you place the /LIBRARY qualifier in a compilation unit, all files in the unit may make reference to modules within that library. Consider the following example:

```
$ CC  ONE + TWO + THREE/LIBRARY
```

Files ONE.C and TWO.C can contain references to modules in THREE.TLB. Consider the following example:

```
$ CC  ONE + TWO + THREE/LIBRARY, FOUR
```

The file FOUR.C cannot contain references to modules in THREE.TLB since FOUR.C is located in a separate compilation unit separated by a comma. The placement of the library file specification does not matter. The following command lines are equivalent:

```
$ CC  THREE/LIBRARY + ONE + TWO
$ CC  ONE + THREE/LIBRARY + TWO
$ CC  ONE + TWO + THREE/LIBRARY
```

/[NO]LINE_DIRECTIVES

Governs whether or not `#line` directives appear in preprocess output files.

The default is /LINE_DIRECTIVES.

/[NO]LIST[=file-spec]

Produces a source program listing. You must specify this qualifier to get a listing. None of the other qualifiers use /LIST by default.

By default, /LIST creates a listing file with the same name as the source file and with a file extension of .LIS. If you include a file specification with the /LIST qualifier, the compiler uses that specification to name the listing file.

In interactive mode, the default is /NOLIST. In batch mode, the default is /LIST. See the descriptions of the qualifiers /[NO]MACHINE_CODE, and /SHOW for related information. (For example, to suppress compiler messages to the terminal or to a batch log file, use the /SHOW=NOTERMINAL qualifier.)

/[NO]MACHINE_CODE[=option]

Lists the generated machine code in the listing file. To produce the listing file, you must also specify /LIST.

On OpenVMS VAX systems, several formats exist to list machine code. *Table 1.11, "/MACHINE_CODE Qualifier Options (VAX only)"* describes the /MACHINE_CODE qualifier options.

Table 1.11. /MACHINE_CODE Qualifier Options *(VAX only)*

Option	Usage
AFTER	Causes the lines of machine code produced during compilation to print after all the source code in the listing.
BEFORE	Causes lines of machine code produced during compilation to print before any source code in the listing.
INTERSPERSED	Produces a listing consisting of lines of source code followed by the corresponding lines of machine code. This is the default option.

On OpenVMS Alpha systems, the format of the generated machine code listing is similar to what you would get using the AFTER keyword on OpenVMS VAX systems.

The default is /NOMACHINE_CODE.

/[NO]MAIN=POSIX_EXIT

Directs the compiler to call `__posix_exit` instead of `exit` when returning from `main`.

The default is /NOMAIN.

/[NO]MEMBER_ALIGNMENT

Controls whether the compiler naturally aligns data structure members. Natural alignment means that data structure members are aligned on the next boundary appropriate to the type of the member, rather than on the next byte. For instance, a `long` variable member is aligned on the next longword boundary; a `short` variable member is aligned on the next word boundary.

Any use of the `#pragma member_alignment` or `#pragma nomember_alignment` directives within the source code overrides the setting established by this qualifier. Specifying /NOMEMBER_ALIGNMENT causes data structure members to be byte-aligned (with the exception of bit-field members).

On OpenVMS Alpha systems, the default is /MEMBER_ALIGNMENT.

On OpenVMS VAX systems, the default is /NOMEMBER_ALIGNMENT.

See the description of `#pragma [no]member_alignment` in *Section 5.4.13, "#pragma [no]member_alignment Directive"*.

/[NO]MMS_DEPENDENCIES [=(option[,…])]

Directs the compiler to produce a dependency file. Dependency files list all source files and included files for each object module. Note that the /OBJECT qualifier has no impact on the dependency file. The dependency file format is:

```
object_file_name :<tab><source file name>)
object_file_name :<tab><full path to first include file>)
object_file_name :<tab><full path to second include file>)
```

Table 1.12, "/MMS_DEPENDENCIES Qualifier Options" shows the /MMS_DEPENDENCIES qualifier options.

Table 1.12. /MMS_DEPENDENCIES Qualifier Options

Option	Usage
FILE[=filespec]	Specifies where to save the dependency file. The default file extension for a dependency file is `.mms`. Other than using this different default extension, /MMS_DEPENDENCY uses the same procedure that the /OBJECT and /LIST qualifiers do for determining the name of the output file.
[NO]SYSTEM_INCLUDE_FILES	Specifies whether or not to include dependency information about system include files (those included with `#include <filename>`.) If omitted, this option defaults to including dependency information about system include files.
TARGET=string	Specifies the target that appears in the output `.mms` file. The default is TARGET="" in which case the target is the source file name with a .OBJ extension, as in previous versions of the compiler. If you specify any string other than .OBJ, that string is used as the target. For the special case of .OBJ, the compiler uses the name of the object file (stripped of any version number and path) for the MMS target. Examples: 1. `$ CC/MMS/OBJ=OUTPUT T.C` This command produces an `.mms` file with a target of T.OBJ: 2. `$ CC/MMS=(TARGET=FOO)/OBJ=OUTPUT T.C` This command produces an `.mms` file with a target of FOO: 3. `$ CC/MMS=(TARGET=.OBJ)/OBJ=OUTPUT T.C` This command produces an `.mms` file with a target of OUTPUT.OBJ:

The default is /NOMMS_DEPENDENCY.

/NAMES=(option1,option2)

Option1 converts all definitions and references of external symbols and psects to the case specified. *Table 1.13, "/NAMES Qualifier Option1 Values"* lists the option1 case values.

Table 1.13. /NAMES Qualifier Option1 Values

Option	Usage
UPPERCASE	Converts to uppercase.
AS_IS	Leaves the case as specified in the source.

Option2 controls whether or not external names greater than 31 characters get truncated or shortened. *Table 1.14, "/NAMES Qualifier Option2 Values"* lists the option2 values.

Table 1.14. /NAMES Qualifier Option2 Values

Option	Usage
/NAMES=TRUNCATED (default)	Truncates long external names.

/NAMES=SHORTENED	Shortens long external names.
	
	A shortened name consists of the first 23 characters of the name followed by a 7-character Cyclic Redundancy Check (CRC) computed by looking at the full name, and then a "$".
	
	The CRC is generated by calling lib$crc as follows:
	
	`long initial_crc = -1;`
	`crc_result = lib$crc(good_crc_table,`
	`&initial_crc,`
	`<descriptor of string to CRC>);`
	
	`where good_crc_table is:`
	
	`/*`
	`** Default CRC table:`
	`**`
	`** This table was taken from Ada's`
	`** generalized name generation algorithm.`
	`** It represents a commonly used CRC`
	`** polynomial known as AUTODIN-II.`
	`** For more information see the VAX`
	`** Macro OpenVMS documentation under the`
	`** CRC VAX instruction.`
	`*/`
	
	`static const unsigned int good_crc_table[16] =`
	`{0x00000000, 0x1DB71064, 0x3B6E20C8, 0x26D930AC,`
	`0x76DC4190, 0x6B6B51F4, 0x4DB26158, 0x5005713C,`
	`0xEDB88320, 0xF00F9344, 0xD6D6A3E8, 0xCB61B38C,`
	`0x9B64C2B0, 0x86D3D2D4, 0xA00AE278, 0xBDBDF21C};`

The default is /NAMES=(UPPERCASE,TRUNCATED), which provides the same conversion-to-uppercase behavior as VAX C, and truncates the name to 31 characters.

Note

On OpenVMS VAX systems, the /NAMES qualifier does not affect the names of the $CODE and $DATA psects.

On OpenVMS Alpha systems, the /NAMES qualifier does not affect the names of the ABS, BSS, $CODE$, $DATA$, $LINK$, $LITERAL$, and $READONLY$ psects.

Specifying /NAMES=SHORTENED turns on the /REPOSITORY qualifier.

/NESTED_INCLUDE_DIRECTORY [=option]

Controls the first step in the compiler's search algorithm for finding files that are included using the quoted form of the `#include` preprocessing directive:

```
#include "file-spec"
```

Table 1.15, "/NESTED_INCLUDE_DIRECTORY Qualifier Options" describes the /NESTED_INCLUDE_DIRECTORY qualifier options.

Table 1.15. /NESTED_INCLUDE_DIRECTORY Qualifier Options

Option	Usage
PRIMARY_FILE	Directs the compiler to search the default file type for headers using the context of the primary source file (the .C file). This means that just the file type (".h" or ".") is used for the default file-spec, but the chain of "related file-specs" used to maintain the sticky defaults for processing the next top-level source file is also applied when searching for the include file. This most closely matches the behavior of VAX C.
INCLUDE_FILE	Directs the compiler to first search the directory of the source file containing the `#include` directive. If the file to be included is not found, the compiler continues searching by following normal inclusion rules.
NONE	Directs the compiler to skip the first step of processing `#include "`*`file`*`.h"` directives. The compiler starts its search for the include file in the /INCLUDE_DIRECTORY directories. It does not start by looking in the directory containing the including file or in the directory containing the top level source file.

The default is /NESTED_INCLUDE_DIRECTORY=INCLUDE_FILE.

/[NO]OBJECT[=file-spec]

Produces an object module. By default, /OBJECT creates an object module file with the same name as that of the first source file of a compilation unit and with the .OBJ file extension. If you include a file specification with /OBJECT, the compiler uses that specification instead.

The compiler executes faster if it does not have to produce an object module. Use the /NOOBJECT qualifier when you need only a listing of a program or when you want the compiler to check a source file for errors. The default is /OBJECT.

Note that the /OBJECT qualifier has no impact on the output file of the /MMS_DEPENDENCIES qualifier.

/[NO]OPTIMIZE[=(option[,…])]

Determines whether VSI C performs code optimizations.

You can specify the options described in *Table 1.16, "/OPTIMIZE Qualifier Options"*.

Table 1.16. /OPTIMIZE Qualifier Options

Option	Usage	
[NO]DISJOINT (VAX only)	Optimizes the generated machine code. For example, the compiler eliminates common subexpressions, removes invariant expressions from loops, collapses arithmetic operations into 3-operand instructions, and places local variables in registers. When debugging VSI C programs, use the /OPTIMIZE=NODISJOINT option if you need minimal optimization; if optimization during debugging is not important, use the /NOOPTIMIZE qualifier.	
[NO]INLINE[=*keyword*]	Provides inline expansion of functions that yield optimized code when they are expanded. You can specify one of the following keywords to control inlining:	
	NONE	No inlining is done, even if requested by the language syntax.
	MANUAL	Inlines only those function calls for which the program explicitly requests inlining.
	AUTOMATIC	Inlines all of the function calls in the MANUAL category, plus additional calls that the compiler determines are appropriate on this platform. On Alpha systems, this is the same as SIZE; on IA-64 systems, this is the same as SPEED. AUTOMATIC is the default.
	SIZE	Inlines all of the function calls in the MANUAL category plus any additional calls that the compiler determines would improve run-time performance without significantly increasing the size of the program.
	SPEED	Performs more aggressive inlining for run-time performance, even when it might significantly increase the size of the program.
	ALL	Inlines every call that can be inlined while still generating correct code. Recursive routines, however, will not cause an infinite loop at compile time. Note that /OPT=INLINE=ALL is not recommended for general use, because it performs very aggressive inlining and can cause the compiler to exhaust virtual memory or take an unacceptably long time to compile.
	The `#pragma noinline` preprocessor directive can be used to prevent inlining of any particular functions under the compiler-selected forms of inlining (SPEED, SIZE, or AUTOMATIC).	

<table>
<tr><th>Option</th><th>Usage</th></tr>
<tr><td></td><td>The <code>#pragma inline</code> preprocessor directive (or the <code>__inline</code> storage-class modifier for OpenVMS Alpha systems) can be used to request inlining of specific functions under the AUTOMATIC or MANUAL forms of inlining.</td></tr>
<tr><td>[NO]INTRINSICS</td><td>Controls whether or not certain functions are handled as intrinsic functions without explicitly enabling each of them as an intrinsic through the <code>#pragma intrinsic</code> preprocessor directive. An intrinsic function is an apparent function call that could be handled as an actual call to the specified function, or could be handled by the compiler in a different manner. By treating the function as an intrinsic, the compiler can often generate faster code. (Contrast with a built-in function, which is an apparent function call that is never handled as an actual function call. There is never a function with the specified name.)

See Section 5.4.10, "#pragma intrinsic Directive " for a list of functions that can be handled as intrinsics.

The /OPTIMZE=INTRINSICS qualifier works together with /OPTIMIZE=LEVEL=n and some other qualifiers to determine how intrinsics are handled:

● If the optimization level specified is less than 4, the intrinsic-function prototypes and call formats are checked, but normal run-time calls are still made.

● If the optimization level is 4 or higher, intrinsic code is generated.

● If /STANDARD=ANSI89 is specified, nonstandard functions are not automatically intrinsic and do not even have their prototypes checked. They are only checked if the nonstandard functions are made intrinsic through <code>#pragma</code> intrinsic.

● Intrinsic code is not generated for math functions that set the errno variable unless /ASSUME=NOMATH_ERRNO is specified. Such math functions, however, do have their prototypes and call formats checked.

The default is /OPTIMIZE=INTRINSICS, which turns on this handling.

To turn it off, specify /NOOPTIMIZE or /OPTIMIZE=NOINTRINSICS, or specify an optimization level less than 4.</td></tr>
<tr><td>LEVEL=n</td><td>Selects the level of optimization. Specify an integer from 0 (no optimization) to 4 (full optimization):

0 — Disables all optimizations. Does not check for unassigned variables.</td></tr>
</table>

Option	Usage	
	1	Includes level 1 optimizations. Enables global optimization. This includes data-flow analysis, code motion, strength reduction and test replacement, split lifetime analysis, and code scheduling.
	2	Includes level 2 optimizations. Enables additional global optimizations that improve speed (at the cost of extra code size), for example: integer multiplication and division expansion (using shifts), loop unrolling, and code replication to eliminate branches.
	3	Includes level 2 optimizations. Enables additional global optimizations that improve speed (at the cost of extra code size), for example: integer multiplication and division expansion (using shifts), loop unrolling, and code replication to eliminate branches.
	4	Includes level 3 optimizations. Enables interprocedural analysis and automatic inlining of small procedures (with heuristics limiting the amount of extra code). This is the default.
	5	Includes level 4 optimizations. Activates software pipelining, which is a specialized form of loop unrolling that in certain cases improves run-time performance. Software pipelining uses instruction scheduling to eliminate instruction stalls within loops, rearranging instructions between different unrolled loop iterations to improve performance. Loops chosen for software pipelining are always innermost loops and do not contain branches or procedure calls. To determine whether using level 5 benefits your particular program, you should time program execution for the same program compiled at levels 4 and 5. For programs that contain loops that exhaust available registers, longer execution times may result with level 5.
[NO]PIPELINE	Controls Activation of the software pipelining optimization. The software pipelining optimization applies instruction scheduling to certain innermost loops, allowing instructions within a loop to "wrap around" and execute in a different iteration of the loop. This can reduce the impact of long-latency operations, resulting in faster loop execution. Software pipelining can be more effective when you combine /OPTIMIZE=PIPELINE with the	

<table>
<tr><th>Option</th><th>Usage</th></tr>
<tr><td></td><td>appropriate /OPTIMIZE=TUNE keyword for the target Alpha processor generation.

Software pipelining also enables the prefetching of data to reduce the impact of cache misses.

Software pipelining is a subset of the optimizations activated by optimization level 5.

To determine whether using /OPTIMIZE=PIPELINE benefits your particular program, you should time program execution for the same program (or subprogram) compiled with and without software pipelining.

For programs containing loops that exhaust available registers, longer execution times can result with optimization level 5, requiring use of /OPTIMIZE=UNROLL=n to limit loop unrolling.</td></tr>
<tr><td>UNROLL=n</td><td>Controls loop unrolling done by the optimizer. UNROLL=n means to unroll loop bodies n times, where n is between 0 and 16. UNROLL=0 means the optimizer will use its own default unroll amount. Specify UNROLL only at level 3 or higher.</td></tr>
<tr><td>TUNE=keyword</td><td>Selects processor-specific instruction tuning for implementations of the Alpha architecture. Regardless of the setting of the /OPTIMIZE=TUNE flag, the generated code will run correctly on all implementations of the Alpha architecture. Tuning for a specific implementation can provide improvements in run-time performance. Code tuned for a specific target might run slower on another target.

You can specify one of the following keywords:
<table>
<tr><td>GENERIC</td><td>Selects instruction tuning that is appropriate for all implementations of the Alpha and Itanium architecture. This option is the default.</td></tr>
<tr><td>HOST</td><td>Selects instruction tuning that is appropriate for the machine on which the code is being compiled.</td></tr>
<tr><td>EV4 (Alpha only)</td><td>Selects instruction tuning for the 21064, 21064A, 21066, and 21068 implementations of the Alpha architecture.</td></tr>
<tr><td>EV5 (Alpha only)</td><td>Selects instruction tuning for the 21164 implementation of the Alpha architecture.</td></tr>
<tr><td>PCA56 (Alpha only)</td><td>Selects instruemction tuning for the 21164PC implementation that uses the byte- and word-manipulation instruction extensions and multimedia instruction extensions of the Alpha architecture.</td></tr>
</table></td></tr>
</table>

Option	Usage	
		Running programs compiled with the PCA56 keyword might incur emulation overhead on EV4, EV5, and EV56 processors, but will still run correctly on OpenVMS Version 7.1 (or higher).
	EV6 (Alpha only)	Selects instruction tuning for the first-generation 21264 implementation of the Alpha architecture.
	EV67 (Alpha only)	Selects instruction tuning for the second-generation 21264 implementation of the Alpha architecture.
	ITANIUM2 (I64 only)	Selects instruction tuning for the Intel Itanium 2 processor.

For OpenVMS VAX systems the default, /OPTIMIZE, is equivalent to /OPTIMIZE=(DISJOINT,INLINE).

For OpenVMS Alpha systems the default, /OPTIMIZE, is equivalent to /OPTIMIZE=(INLINE=AUTOMATIC,LEVEL=4,UNROLL=0,TUNE=GENERIC).

Use /NOOPTIMIZE or /OPTIMIZE=LEVEL=0 for a debugging session to ensure that the debugger has sufficient information to locate errors in the source program.

In most cases, using /OPTIMIZE will make the program execute faster. As a side effect of getting the fastest execution speeds, using /OPTIMIZE can produce larger object modules and longer compile times than /NOOPTIMIZE.

Loop Unrolling

At optimization level 3 or above, VSI C attempts to unroll certain loops to minimize the number of branches and group more instructions together to allow efficient overlapped instruction execution (instruction pipelining). The best candidates for loop unrolling are innermost loops with limited control flow.

As more loops are unrolled, the average size of basic blocks increases. Loop unrolling generates multiple loop code iterations in a manner that allows efficient instruction pipelining.

The loop body is replicated a certain number of times, substituting index expressions. An initialization loop may be created to align the first reference with the main series of loops. A remainder loop may be created for leftover work.

The number of times a loop is unrolled can be determined by the optimizer or the user can specify the limit for loop unrolling using the /OPTIMIZE=UNROLL qualifier. Unless the user specifies a value, the optimizer unrolls a loop 4 times for most loops or 2 times for certain loops (large estimated code size or branches out the loop).

Software Pipelining

Software pipelining and additional software dependence analysis are enabled by using /OPTIMIZE=LEVEL=5, which in certain cases improves run-time performance.

Loop unrolling (enabled at /OPTIMIZE=LEVEL=3 or higher) is constrained in that it cannot schedule across iterations of a loop. Because software pipelining can schedule across loop iterations, it can perform more efficient scheduling that eliminates instruction stalls within loops, by rearranging instructions between different unrolled loop iterations to improve performance.

For example, if software dependence analysis of data flow reveals that certain calculations can be done before or after that iteration of the unrolled loop, software pipelining reschedules those instructions ahead of or behind that loop iteration, at places where their execution can prevent instruction stalls or otherwise improve performance.

Loops chosen for software pipelining:

- Are always innermost loops (those executed the most)
- Do not contain branches or procedure calls

By modifying the unrolled loop and inserting instructions as needed before and/or after the unrolled loop, software pipelining generally improves run-time performance, except for cases where the loops contain a large number of instructions with many existing overlapped operations. In this case, software pipelining may not have enough registers available to effectively improve execution performance, and run-time performance using level 5 may not improve as compared to using level 4.

To determine whether using level 5 benefits your particular program, time program execution for the same program compiled at levels 4 and 5. For programs that contain loops that exhaust available registers, longer execution times may result with level 5.

In cases where performance does not improve, consider compiling using /OPTIMIZE=(UNROLL=1, LEVEL=5) to possibly improve the effects of software pipelining.

`/PDSC_MASK=option`

Forces the compiler to set the PDSC$V_EXCEPTION_MODE field of the procedure descriptor for each function in the compilation unit to the specified value, regardless of the setting of any other qualifiers.

Ordinarily the PDSC$V_EXCEPTION_MODE field gets set automatically by the compiler, depending on the /IEEE_MODE qualifier setting. The /PDSC_MASK qualifier overrides the /IEEE_MODE qualifier setting of this field.

Note

This qualifier is a low-level systems-programming feature that is seldom necessary. Its usage can produce object modules that do not conform to the VMS common language environment and, within C, it can produce nonstandard and seemingly incorrect floating-point behaviors at runtime.

As shown in *Table 1.17, "/PDSC_MASK Qualifier Options"*, the qualifier option keywords are exactly the allowed values defined in the OpenVMS Calling Standard for this field, stripped of the PDSC$V_EXCEPTION_MODE prefix (for example, /PDSC_MASK=SIGNAL sets the field to PDSC$V_EXCEPTION_MODE_SIGNAL).

Table 1.17. /PDSC_MASK Qualifier Options

Option	Maps to	Meaning
SIGNAL	PDSC$K_EXCEPTION_MODE_SIGNAL	Raise exceptions for all except underflow (which is flushed to 0).

Option	Maps to	Meaning
SIGNAL_ALL	PDSC$K_EXCEPTION_MODE_SIGNAL_ALL	Raise exceptions for all.
SILENT	PDSC$K_EXCEPTION_MODE_SILENT	Raise no exceptions. Create only finite values: no infinities, no denorms, no NaNs.
FULL_IEEE	PDSC$K_EXCEPTION_MODE_FULL_IEEE	Raise no exceptions except as controlled by separate IEEE exception-enabling bits. Create exceptional values according to the IEEE standard.
CALLER	PDSC$K_EXCEPTION_MODE_CALLER	Emulate the same mode as the caller. This is useful primarily for writing libraries that can be called from languages other than C.

In the absence of the /PDSC_MASK qualifier, the compiler sets the PDSC$V_EXCEPTION_MODE field automatically, depending on the /IEEE_MODE qualifier setting:

- If /IEEE_MODE is specified with UNDERFLOW_TO_ZERO, DENORM_RESULTS, or INEXACT, then /PDSC_MASK is set to FULL_IEEE.
- In all other cases, /PDSC_MASK is set to SILENT. This setting differs from the calling-standard-specified default of SIGNAL used by FORTRAN, and is largely responsible for the standard-conforming behavior of the math library when called from C or C++ programs.

/[NO]PLUS_LIST_OPTIMIZE

Provides improved optimization and code generation across file boundaries that would not be available if the files were compiled separately.

When you specify /PLUS_LIST_OPTIMIZE on the command line in conjunction with a series of file specifications separated by plus signs, the compiler does not concatenate each of the specified source files together; such concatenation is generally not correct for C code because a C source file defines a scope.

Instead, each file is treated separately for purposes of parsing, except that the compiler issues diagnostics about conflicting external declarations and function definitions that occur in different files. For purposes of code generation, the compiler treats the files as one application and can perform optimizations across the source files.

The default is /NOPLUS_LIST_OPTIMIZE.

/[NO]POINTER_SIZE=option

Controls whether or not pointer-size features are enabled and whether pointers are 32-bits or 64 bits.

The default is /NOPOINTER_SIZE, which disables pointer-size features, such as the ability to use `#pragma pointer_size`, and directs the compiler to assume that all pointers are 32-bit pointers. This default represents no change over previous versions of VSI C.

Table 1.18, "/POINTER_SIZE Qualifier Options" shows the /POINTER_SIZE qualifier options.

Table 1.18. /POINTER_SIZE Qualifier Options

Option	Usage
{SHORT|32}	The compiler assumes 32-bit pointers.
{LONG|64}	The compiler assumes 64-bit pointers.

Specifying /POINTER_SIZE=32 enables pointer-size features and directs the compiler to assume that all pointers are 32-bit pointers.

Specifying /POINTER_SIZE=64 enables pointer-size features and directs the compiler to assume that all pointers are 64-bit pointers.

Specifying the /POINTER_SIZE qualifier enables the following pointer-size features:

- Enables processing of `#pragma pointer_size`.
- Sets the initial default pointer size.
- Predefines the preprocessor macro `__INITIAL_POINTER_SIZE` to 32 or 64.
  If /POINTER_SIZE is omitted from the command line, __INITIAL_POINTER_SIZE is 0, which allows you to use `#ifdef __INITIAL_POINTER_SIZE` to test whether or not the compiler supports 64-bit pointers.
- For /POINTER_SIZE=64, the VSI C RTL name mapping table is changed to select the 64-bit versions of `malloc`, `calloc`, and other RTL routines by default.

For information about other compiler features that affect pointer size or warn about potential pointer size conflicts, see the following:

- /CHECK=POINTER_SIZE
- `#pragma pointer_size`
- `#pragma required_pointer_size`
- `__INITIAL_POINTER_SIZE` predefined macro

The /POINTER_SIZE qualifier must be specified for any program that uses 64-bit pointers.

`/PRECISION[=option]`

Controls whether floating-point operations on `float` variables are performed in single or double precision. *Table 1.19, "/PRECISION Qualifier Options"* shows the /PRECISION qualifier options.

Table 1.19. /PRECISION Qualifier Options

Option	Usage
SINGLE	Performs floating-point operations in single precision.
DOUBLE	Performs floating-point operations in double precision.

Your code may execute faster if it contains `float` variables and is compiled with /PRECISION=SINGLE. However, the results of your floating-point operations will be less precise. See the *VSI C Reference Manual* [https://docs.vmssoftware.com/vsi-c-language-reference-manual/] for more information on floating-point variables.

The default is /PRECISION=DOUBLE for /STANDARD=VAXC and /STANDARD=COMMON compiler modes.

The default is /PRECISION=SINGLE for /STANDARD=ANSI89 and /STANDARD=RELAXED compiler modes.

/[NO]PREFIX_LIBRARY_ ENTRIES[=(option[,…])]

The VSI C Run-Time Library (RTL) shareable image, DECC$SHR.EXE, resides in SYS$LIBRARY with a DECC$ prefix for its entry points. The linker searches IMAGELIB.OLB to locate the shareable image. Every external name in IMAGELIB.OLB has a DECC$ prefix, and, therefore, has an OpenVMS conformant name space (a requirement for inclusion in IMAGELIB).

The /[NO]PREFIX_LIBRARY_ENTRIES qualifier lets you control the VSI C RTL name prefixing. *Table 1.20, "/PREFIX_LIBRARY_ENTRIES Qualifier Options"* describes the /PREFIX_LIBRARY_ENTRIES qualifier options.

Table 1.20. /PREFIX_LIBRARY_ENTRIES Qualifier Options

Option	Usage
EXCEPT = (*name*,...)	The names specified are not prefixed.
ALL_ENTRIES	All VSI C RTL names, as well as C99 names not supported by the underlying C RTL, are prefixed.
ANSI_C89_ENTRIES	Only C Standard 89 (C89) library names are prefixed.
C99_ENTRIES	Only C Standard 99 (C99) library names are prefixed. These are a superset of the external names prefixed under /PREFIX=ANSI_C89_ENTRIES and a subset of those prefixed under /PREFIX=ALL_ENTRIES. The compiler will prefix C99 entries based on their inclusion in the standard, not on the availability of their implementations in the run-time library. So calling functions introduced in C99 that are not yet implemented in the C RTL will produce unresolved references to symbols prefixed by DECC$ when the program is linked. In addition, the compiler will issue a CC-W-NOTINCRTL message when it prefixes a name that is not in the current C RTL.
RTL="*name*"	Generates references to the C RTL indicated by the *name* keyword. (The *name* keyword has a length limit of 24 characters for OpenVMS VAX systems and 1017 characters for OpenVMS Alpha systems.) If no keyword is specified, then references to the VSI C RTL are generated by default. To use an alternate RTL, see its documentation for the name to use.

If you want no names prefixed, specify /NOPREFIX_LIBRARY_ENTRIES.

For /STANDARD=ANSI89, the default is /PREFIX=ANSI_C89_ENTRIES.

For /STANDARD=C99, the default is /PREFIX=C99_ENTRIES.

For all other compiler modes, the default is /PREFIX=ALL.

/[NO]PREPROCESS_ONLY [=filename]

Gives the same functionality as the `-E` qualifier on UNIX C compilers. When specified, it performs only the actions of the preprocessor phase and writes the resulting processed text to a file. No semantic or

syntax processing is done. Furthermore, no object file, diagnostic file, listing file, or analysis data file is produced.

If you do not specify a file name for the preprocessor output, the name of the output file defaults to the file name of the input file with a .I file type.

The default is /NOPREPROCESS_ONLY.

/[NO]PROTOTYPE [=(option[,…])]

Creates an output file containing function prototypes for all global functions defined in the module being compiled.

Standard-style prototypes are created even for functions defined with Kernighan and Ritchie style syntax.

This qualifier can be used to convert to Standard-sytle prototypes or just to ensure that every function definition has a compatible explicit declaration, thereby avoiding implicit declarations that can sometimes produce surprising results.

Table 1.21, "/PROTOTYPE Qualifier Options" describes the /PROTOTYPE qualifier options.

Table 1.21. /PROTOTYPE Qualifier Options

Option	Usage
[NO]IDENTIFIERS	Indicates that identifier names are to be included in the prototype declarations that appear in the output file. The default is NOIDENTIFIERS.
[NO]STATIC_FUNCTIONS	Indicates that prototypes for static function definitions are to be included in the output file. The default is NOSTATIC_FUNCTIONS.
FILE=*filename*	Specifies the output file name. When not specified, the output file name has the same defaults as the listing file, except that the file extension is .CH instead of .LIS.

The default is /NOPROTOTYPES.

/PSECT_MODEL= [NO]MULTILANGUAGE

Controls whether the compiler allocates the size of overlaid psects to ensure compatibility when the psect is shared by code created by other VSI compilers.

The problem this switch solves can occur when a psect generated by a FORTRAN COMMON block is overlaid with a psect consisting of a C struct. Because FORTRAN COMMON blocks are not padded, if the C struct is padded, the inconsistent psect sizes can cause linker error messages.

Compiling with /PSECT_MODEL=MULTILANGUAGE ensures that VSI C uses a consistent psect size allocation scheme. The corresponding FORTRAN switch is /ALIGN=COMMON=[NO]MULTILANGUAGE.

The default is /PSECT=NOMULTILANGUAGE, which is the old default behavior of the compiler, and is sufficient for most applications.

/REENTRANCY=option

Controls the type of reentrancy that reentrant VSI C RTL routines will exhibit. (See the `decc $set_reentrancy` RTL routine.)

This qualifier is for use only with a module containing the `main` routine.

The reentrancy level is set at runtime according to the /REENTRANCY qualifier specified while compiling the module containing the `main` routine.

Table 1.22, "/REENTRANCY Qualifier Options" describes the /REENTRANCY qualifier options.

Table 1.22. /REENTRANCY Qualifier Options

Option	Usage
AST	Uses the __TESTBITSSI built-in function to perform simple locking around critical sections of RTL code, and may additionally disable asynchronous system traps (ASTs) in locked region of codes. This type of locking should be used when AST code contains calls to VSI C RTL I/O routines.
MULTITHREAD	Designed to be used in conjunction with the DECthreads product. It performs DECthreads locking and never disables ASTs.
NONE	Gives optimal performance in the RTL, but does absolutely no locking around critical sections of RTL code. It should only be used in a single threaded environment when there is no chance that the thread of execution will be interrupted by an AST that would call the VSI C RTL.
TOLERANT	Uses the __TESTBITSSI built-in function to perform simple locking around critical sections of RTL code, but ASTs are not disabled. This type of locking should be used when ASTs are used and must be delivered immediately.

The default is /REENTRANCY=TOLERANT.

`/REPOSITORY=option`

Specifies a repository for the compiler to store shortened external name information. When /NAMES=SHORTENED is specified, the compiler stores to the repository any external names that were shortened. The demangler utility can then be used to map the shortened names back to the names used in the original C program.

By default, the qualifier is not active unless /NAMES=SHORTENED has been specified, in which case the default is /REPOSITORY=[.CXX_REPOSITORY].

The default name of the repository is the same as that used by the VSI C++ compiler for decoding mangled names. This is intentional. A C++ mangled name cannot match a shortened name, so a single repository can be used by both the VSI C and VSI C++ compilers.

`/ROUNDING_MODE=option`

If /FLOAT=IEEE_MODE is specified, the /ROUNDING_MODE qualifier lets you select one of the following IEEE rounding modes:

Option	Usage
NEAREST	Sets the normal rounding mode (unbiased round to nearest). This is the default.
DYNAMIC	Sets the rounding mode for IEEE floating-point instructions dynamically, as determined from the contents of the floating-point control register.
MINUS_INFINITY	Rounds toward minus infinity.

Option	Usage
CHOPPED	Rounds toward 0.

If /FLOAT=G_FLOAT or /FLOAT=D_FLOAT is specified, then rounding defaults to /ROUNDING_MODE=NEAREST, with no other choice of rounding mode.

/[NO]SHARE_GLOBALS

Controls whether the compiler will treat declarations of objects with the `globaldef` keyword as shared or not shared.

Also, in conjunction with the /EXTERN_MODEL qualifier, controls whether the initial `extern_model` is shared or not shared (for those `extern_model`s where it is allowed). The initial `extern_model` of the compiler is a fictitious pragma constructed from the settings of the /EXTERN_MODEL and /SHARE_GLOBALS qualifiers.

The default value is /NOSHARE_GLOBALS. This default value is different from VAX C, which treats external objects as shared by default. As a result, you may experience the following impact:

- Linking old object files or object libraries with newly produced object files might generate “conflicting attributes for psect” messages. As long as you are not building shareable libraries, you can safely ignore these messages.
- Building shareable libraries will be easier.
- On OpenVMS VAX systems, when linking external symbols against FORTRAN common blocks, you should specify /SHARE_GLOBALS to suppress “conflicting attributes for psect” messages; although they can otherwise be ignored. (VAX only)

/SHOW[=(option[,…])]

Sets or cancels listing options. You must use the /LIST qualifier with the /SHOW qualifier to use any of the /SHOW options. *Table 1.23, "/SHOW Qualifier Options"* describes the /SHOW qualifier options.

Table 1.23. /SHOW Qualifier Options

Option	Usage
ALL	Prints all listing information.
[NO]BRIEF	Creates the same listing as the option SYMBOLS except that BRIEF eliminates from the list any identifiers that are not referenced in the program, and are not members of a structure or union that is referenced in the program. The NOBRIEF option is the default.
[NO]CROSS_REFERENCE	Specifies whether the compiler generates cross-references. If you specify /SHOW=CROSS_REFERENCE, the compiler lists, for each variable referenced in the procedure, the line numbers of the lines on which the variable is referenced. You may use /SHOW=CROSS_REFERENCE with /SHOW=SYMBOLS. Otherwise, specifying /SHOW=CROSS_REFERENCE also gives you /SHOW=BRIEF. To obtain any type of listing, you must specify /LIST.

Option	Usage
	Specifying /SHOW=[NO]CROSS_REFERENCE is the same as specifying /[NO]CROSS_REFERENCE. The NOCROSS_REFERENCE option is the default.
[NO]DICTIONARY	Places CDD/Repository definitions—included in the program with the `#pragma dictionary` preprocessor directive —into the listing file. These data definitions are marked in the listing file with an uppercase letter D in the listing margin. The NODICTIONARY option is the default.
[NO]EXPANSION	Places final macro expansions in the program listing. However, expansion text for preprocessing directives is not shown. When you specify this option, the number printed in the margin indicates the maximum depth of macro substitutions that occur on each line. The NOEXPANSION option is the default.
[NO]HEADER	Produces the header lines at the top of each page of a listing. The HEADER option is the default.
[NO]INCLUDE	Places the contents of `#include` files and modules in the program listing. The NOINCLUDE option is the default.
[NO]INTERMEDIATE (VAX only)	Places all intermediate and final macro expansions in the program listing. The NOINTERMEDIATE option is the default.
[NO]MESSAGES	Lists all messages that are in effect at compilation (based on the settings of /STANDARD, /WARNINGS, and #pragma message). The NOMESSAGE option is the default.
NONE	Creates an empty listing file with only the header. If you specify this option on a CC command line that contains /LIST and /MACHINE_CODE, the compiler places machine code in the listing file.
[NO]SOURCE	Places the source program statements in the program listing. The SOURCE option is the default.
[NO]STATISTICS	Places compiler performance statistics in the program listing. The NOSTATISTICS option is the default.
[NO]SYMBOLS	Places the symbol table of the compiled program in the program listing. The symbol table includes a list of all functions, the sizes and attributes of all variables referenced

Option	Usage
	in the program, and a program section summary and function definition map. The NOSYMBOLS option is the default.
[NO]TERMINAL (VAX only)	Displays compiler messages to the terminal. Use /SHOW=NOTERMINAL to suppress compiler messages to the terminal or to a batch log file. The TERMINAL option is the default.
[NO]TRANSLATION (VAX only)	Places into the listing file all UNIX system file specifications that the compiler translates to OpenVMS file specifications. See the *VSI C Run-Time Library Reference Manual for OpenVMS Systems* [https://docs.vmssoftware.com/vsi-c-run-time-library-reference-manual-for-openvms-systems/] for more information on file translation. The NOTRANSLATION option is the default.

/[NO]STANDARD[=(option[,…])]

Defines the compilation mode, directing the compiler to flag certain VSI C-specific constructs and VSI C relaxations of conventional C language constructs and rules. For example, the conversions from pointer to integer and back again are subject to more stringent tests when you specify /STANDARD=ANSI89.

Table 1.24, "/STANDARD Qualifier Options" describes the /STANDARD qualifier options.

Table 1.24. /STANDARD Qualifier Options

Option	Usage
ANSI89	Places the compiler in strict C Standard mode.
C99	Places the compiler in strict ISO/IEC C99 Standard mode. Note that /STANDARD=C99 is not fully supported on VAX systems. Specifying /STANDARD=C99 on OpenVMS VAX systems produces a warning and puts the compiler into /STANDARD=RELAXED mode.
LATEST	Places the compiler in the latest ISO C standard dialect. /STANDARD=LATEST is currently equivalent to /STANDARD=C99, but is subject to change when newer versions of the ISO C standard are released.
RELAXED	Places the compiler in relaxed C Standard mode.
MS	Interprets source programs according to certain language rules followed by Microsoft's Visual C++ compiler.
ISOC94	Places the compiler in ISO C 94 mode, which enables digraph processing and defines the macro `__STDC_VERSION__=199409L`. Digraphs are pairs of characters that translate into a single character, much like trigraphs, except that trigraphs get replaced inside string literals, but digraphs do not. The digraphs are:

Digraph	Character Represented
<:	[

Option	Usage	
	:>	]
	<%	{
	%>	}
	%:	#
	%:%:	##
	The ISOC94 option can be specified alone or in combination with any other option except VAXC. If specified alone, ISOC94 provides a default major mode of RELAXED.	
COMMON	Places the compiler in common C mode. This mode enforces K & R programming style; that is, compatibility with older UNIX compilers such as `pcc` and `gcc`.	
VAXC	Places the compiler in VAX C mode.	
PORTABLE	Places the compiler in RELAXED mode, and enables the issuance of diagnostics that warn about any nonportable usages encountered. /STANDARD=PORTABLE is supported for VAX C compatibility only. It is equivalent to the recommended combination of qualifiers /STANDARD=RELAXED/WARNINGS=ENABLE=PORTABLE.	
MIA	Places the compiler in strict C Standard mode with some behavior differences, as required by the MIA standard: ● On OpenVMS VAX systems, G_floating becomes the default floating-point format for `double` variables. (VAX only) On OpenVMS Alpha systems, G_floating is the default in any case. (Alpha only) ● In structures, zero-length bit fields cause the next bit field to start on an integer boundary, rather than on a character boundary. Compiling a program with /STANDARD=MIA sets the `__MIA` predefined macro to 1.	

The default is /NOSTANDARD, which is equivalent to /STANDARD=RELAXED.

If you specify /STANDARD, you must supply at least one option.

With one exception, the /STANDARD qualifier options are mutually exclusive. Do not combine them. The exception is that you can specify /STANDARD=ISOC94 with any other option except VAXC.

VSI C modules compiled in different modes can be linked and executed together.

Also see the `__HIDE_FORBIDDEN_NAMES` predefined macro (*Section 6.1.7, "The __HIDE_FORBIDDEN_NAMES Macro"*).

/[NO]TIE

Enables the compiled code to be used in combination with translated images, either because the code might call into a translated image or might be called from a translated image. The default is /NOTIE.

/[NO]UNDEFINE=(identifier[,…])

See /[NO]DEFINE in this section.

/[NO]UNSIGNED_CHAR

By default, `char` is a signed character type. The /UNSIGNED_CHAR qualifier lets you change this default to an unsigned character type, which causes all plain `char` declarations to have the same representation and set of values as `unsigned char` declarations. The default is /NOUNSIGNED_CHAR.

/VAXC (VAX only)

Invokes the VAX C compiler.

The CC command is used to invoke either the VAX C or VSI C compiler. If your system has a VAX C compiler installed on it, the VSI C installation procedure provides the option of specifying which compiler will be invoked by default when just the CC command is used. To invoke the compiler that is not the default, use the CC command with the appropriate qualifier: CC/DECC for the VSI C compiler, or CC/VAXC for the VAX C compiler.

If your system does not have a VAX C compiler installed on it, the CC command will invoke the VSI C compiler, and the /VAXC qualifier is not supported.

/[NO]VERSION

Directs the compiler to print out the compiler version and platform. The compiler version is the same as in the listing file.

This qualifier makes it easier for you to report what compiler you are using.

Note

To display the compiler version and platform when issuing the CC command for a source file that does not exist, enter:

```
CC/DECC/VERSION NL:
```

/[NO]WARNINGS[=(option[,…])]

Controls the issuance of compiler diagnostic messages or groups of messages. It also allows for the severity of messages to be modified. The default qualifier, /WARNINGS, enables all warning and informational messages for the compiler mode you are using. The /NOWARNINGS qualifier suppresses the warning and informational messages. Also see the `#pragma message` preprocessor directive.

Table 1.25, "/WARNINGS Qualifier Options" describes the /WARNING qualifier options.

For a description of what to specify for the *message-list*, see the `#pragma message` preprocessor directive (*Section 5.4.14, "#pragma message Directive"*).

Table 1.25. /WARNINGS Qualifier Options

Option	Usage
DISABLE=*message-list*	Suppresses the issuance of the specified messages.

Option	Usage
	Only messages of severity Warning (W) or Information (I) can be disabled. If the message has severity of Error (E) or Fatal (F), it is issued regardless of any attempt to disable it.
ENABLE=*message-list*	Enables issuance of the specified messages.
NOINFORMATIONALS	Suppresses informational messages.
EMIT_ONCE=*message-list*	Emits the specified messages only once per compilation. Certain messages are emitted only the first time the compiler encounters the causal condition. When the compiler encounters the same condition later in the program, no message is emitted. Messages about the use of language extensions are an example of this kind of message. To emit one of these messages every time the causal condition is encountered, use the EMIT_ALWAYS option. Errors and Fatals are always emitted. You cannot set them to EMIT_ONCE.
EMIT_ALWAYS=*message-list*	Emits the specified messages at every occurrence of the causal condition.
ERRORS=*message-list*	Sets the severity of the specified messages to Error. Supplied Error messages and Fatal messages cannot be made less severe. (Exception: A message can be upgraded from Error to Fatal, then later downgraded to Error again, but it can never be downgraded from Error.) Warnings and Informationals can be made any severity.
FATALS=*message-list*	Sets the severity of the specified messages to Fatal.
INFORMATIONALS=*message-list*	Sets the severity of the specified messages to Informational. Note that Fatal and Error messages cannot be made less severe.
WARNINGS=*message-list*	Sets the severity of the specified messages to Warning. Note that Fatal and Error messages cannot be made less severe.
VERBOSE	Displays the full message information for every compiler message encountered. This information includes the message description and user action, as well as the identifier, severity, and message text. When /WARNINGS=VERBOSE is used with /LIST/SHOW=MESSAGES, a list of all messages in effect at compilation are added to the listing file, showing the full information for each message.

Note

- If a message is on both the enabled and disabled list, it is disabled.

- If a message is on both the EMIT_ONCE and the EMIT_ALWAYS list, it is considered to be on the EMIT_ONCE list.
- If a message is on more than one of the FATALS, ERRORS, WARNINGS, or INFORMATIONALS lists, the message is given the least severe level.
- The NOINFORMATIONALS option is not the negation of INFORMATIONALS=msg-list. It is valid to specify:

  ```
  /WARNINGS=(INFORMATIONALS=message_list,NOINFORMATIONALS)
  ```

  This has the effect of making the messages on the message_list informationals, and causing the compiler to suppress any informational messages.
- One of the message groups described in the `#pragma message` description in *Section 5.4.14, "#pragma message Directive"* is UNUSED, which enables messages that report apparently unnecessary `#include` files and CDD records.

  However, unlike any other messages, these messages must be enabled on the command line (/WARNINGS=ENABLE=UNUSED) to be effective. Any `#pragma message` directives within the source have no effect on these messages; their state is determined only by processing the command line.

The default is /WARNINGS=ENABLE=LEVEL3.

1.3.5. Compiler Diagnostic Messages

If there are errors in your source file when you compile your program, the VSI C compiler signals these errors and displays diagnostic messages. Reference the message, locate the error, and, if necessary, correct the error. See *Appendix D, "VSI C Compiler Messages"* or the online help for a description of all compiler diagnostic messages.

You can control the issuance of specific compiler diagnostic messages or groups of messages with the /[NO]WARNINGS command-line qualifier (*Section 1.3.4, "CC Command Qualifiers"*) and the `#pragma message` preprocessor directive (*Section 5.4.14, "#pragma message Directive"*).

To display a particular compiler diagnostic message online, enter the following command:

```
$ HELP CC/DECC MESSAGE mnemonic     (VAX only)
$ HELP CC MESSAGE mnemonic     (Alpha, I64)
```

To display a list of all message mnemonics, enter the following command:

```
$ HELP /DECC MESSAGE     (VAX only)
$ HELP CC MESSAGE     (Alpha, I64)
```

Diagnostic messages have the following format:

```
%CC-s-ident, message-text
                Listing line number m
                At line number n in name
```

%CC

The facility or program name of the VSI C compiler. This portion indicates that the message is being issued by VSI C.

s

The severity of the error, represented in the following way:

F	Fatal error. The compiler stops executing when a fatal error occurs and does not produce an object module. You must correct the error before you can compile the program.
E	Error. The compiler continues, but does not produce an object module. You must correct the error before you can successfully compile the program.
W	Warning. The compiler produces an object module. It attempts to correct the error in the statement, but you should verify that the compiler's action is acceptable. Otherwise, your program may produce unexpected results.
I	Information. This message usually appears with other messages to inform you of specific actions taken by the compiler. No action is necessary on your part.

ident
The message identification. This is a descriptive abbreviation (mnemonic) of the message text.

message-text
The compiler's message. In many cases, it consists of more than one line of output. A message generally provides you with enough information to determine the cause of the error so that you can correct it.

Listing line number *m*
The integer *m*, which gives you the line number in the listing file where the error occurs. This information is given when you specify the /LIST qualifier.

At line number *n* in *name*
The integer *n*, which gives you the number of the line where the error occurs. The number is relative to the beginning of the file or text library module specified by *name*. You can use the `#line` directive to change both the line number and name that appear in the message.

1.4. Linking a VSI C Program

After you compile a VSI C source program or module, use the DCL command LINK to combine your object modules into one executable image, which can then be executed by the OpenVMS system. A source program or module cannot run on the OpenVMS system until it is linked.

When you execute the LINK command, the linker performs the following functions:

- Resolves local and global symbolic references in the object code
- Assigns values to the global symbolic references
- Signals an error message for any unresolved symbolic reference
- Allocates virtual memory space for the executable image

When using the LINK command on development systems, use the /DEBUG qualifier to link your program module. The /DEBUG qualifier appends to the image all the symbol and line number information appended to the object modules plus information on global symbols, and causes the image to run under debugger control when it is executed.

The LINK command produces an executable image by default. However, you can also use the LINK command to obtain shareable images and system images. The /SHAREABLE qualifier directs the linker

to produce a shareable image; the /SYSTEM qualifier directs the linker to produce a system image. See *Section 1.4.2, "LINK Command Qualifiers"* for a complete description of these and other LINK command qualifiers.

For a complete discussion of the OpenVMS Linker, see the *VSI OpenVMS Linker Utility Manual*.

1.4.1. The LINK Command

The LINK command has the following format:

```
LINK[/command-qualifier]... {file-spec[/file-qualifier...]},...
```

/command-qualifier...

Output file options.

file-spec

The input files to be linked.

/file-qualifier...

Input file options.

If you specify more than one input file, you must separate the input file specifications with a plus sign (+) or a comma (,).

By default, the linker creates an output file with the name of the first input file specified and the file type EXE. If you link more than one file, you should specify the file containing the main program first. Then, the name of your output file will have the same name as your main program module.

The execution of a program will begin at the function whose identifier is `main`, or, if there is no function with this identifier, at the first function seen by the VMS linker.

Note

Unexpected results might occur if you don't have a function called `main`.

The following command line links the object files MAINPROG.OBJ, SUBPROG1.OBJ, and SUBPROG2.OBJ to produce one executable image called MAINPROG.EXE:

```
$ LINK MAINPROG.OBJ, SUBPROG1.OBJ, SUBPROG2.OBJ
```

Note

Unlike VAX C, VSI C does not require you to define any LNK$LIBRARY logicals.

1.4.2. LINK Command Qualifiers

You can use the LINK command qualifiers to modify the linker's output, as well as to invoke the debugging and traceback facilities. Linker output consists of an image file and an optional map file.

The following list summarizes some of the most commonly used LINK command qualifiers. A brief description of each qualifier follows this list. For a complete list of LINK qualifiers, see the *VSI OpenVMS Linker Utility Manual*.

Command Qualifiers	Default
/BRIEF	None.
/[NO]CROSS_REFERENCE	/NOCROSS_REFERENCE
/[NO]DEBUG	/NODEBUG
/[NO]EXECUTABLE[=file-spec]	/EXECUTABLE=name.EXE
/FULL	None
/[NO]MAP	/MAP (batch) /NOMAP (interactive)
/[NO]SHAREABLE[=file-spec]	/[NO]SHAREABLE[=file-spec]
/[NO]TRACEBACK	/TRACEBACK

/BRIEF

Produces a summary of the image's characteristics and a list of contributing modules. This qualifier is mutually exclusive with /FULL.

/[NO]CROSS_REFERENCE

Produces cross-reference information for global symbols; /NOCROSS_REFERENCE suppresses cross-reference information. The default is /NOCROSS_REFERENCE.

/[NO]DEBUG

Includes the OpenVMS Debugger in the executable image and generates a symbol table; /NODEBUG causes the linker to prevent debugger control of the program. The default is /NODEBUG.

/[NO]EXECUTABLE [=file-spec]

Produces an executable image. /NOEXECUTABLE suppresses production of an image file. The default is /EXECUTABLE.

/FULL

Produces a summary of the image's characteristics, a list of contributing modules, listings of global symbols by name and by value, and a summary of characteristics of image sections in the linked image. This qualifier is mutually exclusive with /BRIEF.

/[NO]MAP

Generates a map file; /NOMAP suppresses the map. The default is /MAP in batch mode and /NOMAP in interactive mode.

/[NO]SHAREABLE[=file-spec]

Creates a shareable image. /NOSHAREABLE generates an executable image. The default is /NOSHAREABLE.

/[NO]TRACEBACK

Generates symbolic traceback information when error messages are produced; NOTRACEBACK suppresses traceback information. The default is /TRACEBACK.

1.4.3. Linker Input Files

You can specify the object modules to be included in an executable image in any of the following ways:

- Specify input file specifications for the object modules.

  If no file type is specified, the linker searches for an object file with the file type OBJ.

- Specify one or more object module library files.

  You can specify either the name of an object module library with the /LIBRARY qualifier or the names of the object modules contained in an object module library with the /INCLUDE qualifier. *Section 1.4.6, "Object Module Libraries"* describes the uses of object module libraries.

- Specify an options file.

  An options file can contain additional file specifications for the LINK command, as well as special linker options. You must use the /OPTIONS qualifier to specify an options file. For more information on options files, see the *VSI OpenVMS Linker Utility Manual*.

Table 1.26, "OpenVMS Linker Default File Types for Input Files" shows the default input file types for the linker.

Table 1.26. OpenVMS Linker Default File Types for Input Files

File Type	File
OBJ	Object module
OLB	Library
OPT	Options file

1.4.4. Linker Output Files

When you enter the LINK command interactively and do not specify any qualifiers, the linker creates only an executable image file. By default, the resulting image file has the same file name as that of the first object module specified with a file type of EXE.

In a batch job, the linker creates both an executable image file and storage map file by default. The default file type for map files is MAP.

To specify an alternative name for a map file or image file or to specify an alternative output directory or device, you can include a file specification on the /MAP or /EXECUTABLE qualifier. In the following example, the LINK command creates the image file [PROJECT.EXE]UPDATE.EXE and the map file [PROJECT.MAP]UPDATE.MAP:

```
$ LINK UPDATE/EXECUTABLE=[PROJECT.EXE]/MAP=[PROJECT.MAP]
```

1.4.5. Linking Against Object Module Libraries and Shareable Images

Linking against object modules (stored in object module libraries) or against shareable images are ways of allowing your program to access data and routines outside of your compilation units. You can either create the object module libraries and the shareable images or use the ones provided by VSI. To access

data in object modules and shareable images, you can use LINK command qualifiers, OpenVMS logical names, and options files. For more information about object module libraries, see the *VSI OpenVMS Linker Utility Manual*.

The VSI C Run-Time Library (RTL) for OpenVMS systems also provides two formats for you to choose from: shareable images or object module libraries. Depending on which type of RTL you want to use and on which type of functions you plan on calling from your programs, you need to supply information to the linker that specifies which versions of the functions to access.

When you use the C RTL and its corresponding header files, remember that the C RTL ships with the OpenVMS operating system and the header files ship with the VSI C compiler. Since the releases of the compiler and of the operating system are not synchronized, there may be compatibility issues that you need to consider to use the RTL properly. See the VSI C release notes (by entering HELP CC/DECC RELEASE_NOTES on the DCL command line) for information that may pertain to this issue.

For a description of the various ways to link with the C RTL, see the *VSI C Run-Time Library Reference Manual for OpenVMS Systems* [https://docs.vmssoftware.com/vsi-c-run-time-library-reference-manual-for-openvms-systems/].

1.4.6. Object Module Libraries

You can make program modules accessible to other users by storing them in an object module library. To link modules contained in an object module library, use the /INCLUDE qualifier and specify the modules you want to link. The following example links the subprogram modules EGGPLANT, TOMATO, BROCCOLI, and ONION with the main program module GARDEN:

```
$ LINK GARDEN, VEGGIES/INCLUDE=(EGGPLANT,TOMATO,BROCCOLI,ONION)
```

An object module library can also contain a symbol table with the names of each global symbol in the library, and the name of the module in which they are defined. You specify the name of the object module library containing symbol definitions with the /LIBRARY qualifier. When you use the /LIBRARY qualifier during a linking operation, the linker searches the specified library for all unresolved references found in the included modules during compilation.

The following example uses the library RACQUETS to resolve undefined symbols in BADMINTON, TENNIS, and RACQUETBALL:

```
$ LINK BADMINTON, TENNIS, RACQUETBALL, RACQUETS/LIBRARY
```

You can define an object module library to be your default library by using the DCL command DEFINE LNK$LIBRARY. The linker searches default user libraries for unresolved references after it searches modules and libraries specified in the LINK command. For more information about the DEFINE command, see the *VSI OpenVMS DCL Dictionary*.

For more information about object module libraries, see the *VSI OpenVMS Linker Utility Manual*.

1.4.7. Linker Error Messages

If the linker detects any errors while linking object modules, it displays messages indicating the cause and severity of the error. If any error or fatal error conditions occur (that is, errors with severities of E or F), the linker does not produce an image file.

The messages produced by the linker are descriptive, and you do not usually need additional information to determine the specific error. Some common errors that occur during linking are as follows:

- An object module has compilation errors.

  This occurs when you try to link a module that produced warning messages during compilation. You can usually link compiled modules for which the compiler generated messages, but verify that the modules will produce the output you expect.

- The input file has a file type other than OBJ and no file type was specified on the command line.

  If you do not specify a file type, the linker searches for a file that has a file type of OBJ by default. If the file is not an object file and you do not identify it with the appropriate file type, the linker signals an error message and does not produce an image file.

- You tried to link a nonexistent module.

  The linker signals an error message if you misspell a module name on the command line or if the compilation contains fatal diagnostics.

- A reference to a symbol name remains unresolved.

  An error occurs when you omit required module or library names from the command line and the linker cannot locate the definition for a specified global symbol reference. Consider, for example, the following LINK command for a main program module, OCEAN.OBJ, that calls the subprogram modules REEF.OBJ, SHELLS.OBJ, and SEAWEED.OBJ:

  ```
  $ LINK OCEAN, REEF, SHELLS
  ```

  Because SEAWEED is not linked, the linker issues the following error messages:

  ```
  %LINK-W-NUDFSYMS, 1 undefined symbol
  %LINK-I-UDFSYMS,        SEAWEED
  %LINK-W-USEUNDEF, module "OCEAN" references undefined symbol "SEAWEED"
  %LINK-W-DIAGISUED, completed but with diagnostics
  ```

If an error occurs when you link modules, you can often correct it by reentering the command and specifying the correct modules or libraries. If an error indicates that a program module cannot be located, you may be linking the program with the wrong RTL.

For a complete list of linker messages, see the *OpenVMS System Messages and Recovery Procedures Reference Manual*.

1.5. Running a VSI C Program

After you link your program, you can use the DCL command RUN to execute it. The RUN command has the following format:

```
RUN [/[NO]DEBUG] file-spec [/[NO]DEBUG]
```

/[NO]DEBUG

An optional qualifier. Specify the /DEBUG qualifier to invoke the debugger if the image was not linked with it. You cannot use /DEBUG on images linked with the /NOTRACEBACK qualifier. If the image was linked with the /DEBUG qualifier and you do not want the debugger to prompt you, use the /NODEBUG qualifier. The default action depends on whether the file was linked with the /DEBUG qualifier.

file-spec

The file you want to run.

The execution of a program begins at the function whose identifier is `main`, or, if there is no function with this identifier, at the first function seen by the VMS linker.

Note

Unexpected results might occur if you don't have a function called `main`.

The following example executes the image SAMPLE.EXE without invoking the debugger:

```
$ RUN SAMPLE/NODEBUG
```

For more information on debugging programs, see *Section C.1, "OpenVMS Debugger"*.

During execution, an image can generate a fatal error called an *exception condition*. When an exception condition occurs, the system displays an error message. Run-time errors can also be issued by the operating system or by utilities.

When an error occurs during the execution of a program, the program is terminated and the OpenVMS condition handler displays one or more messages on the currently defined SYS$ERROR device.

A message is followed by a traceback. For each module in the image that has traceback information, the condition handler lists the modules that were active when the error occurred, which shows the sequence in which the modules were called.

For example, if an integer divide-by-zero condition occurs, a run-time message like the following appears:

```
%SYSTEM-F-INTDIV, arithmetic trap, integer divide by zero
    at PC=00000FC3, PSL=03C00002
```

This message is followed by a traceback message similar to the following:

```
%TRACE-F-TRACEBACK, symbolic stack dump follows

module name   routine name   line       rel  PC         abs PC
A             C                 8      00000007      00000FC3
B             main           1408      000002F7      00000B17
```

The information in the traceback message follows:

module name

The name or names of an image module that was active when the error occurred.

The first module name is that of the module in which the error occurred. Each subsequent line gives the name of the caller of the module named on the previous line. In this example, the modules are A and B; main called C.

routine name

The name of the function in the calling sequence.

line

The compiler-generated line number of the statement in the source program where the error occurred, or at which the call or reference to the next procedure was made. Line numbers in these messages match those in the listing file (not the source file).

rel PC

The value of the PC (program counter). This value represents the location in the program image at which the error occurred or at which a procedure was called. The location is relative to the virtual memory address that the linker assigned to the code program section of the module indicated by module name.

abs PC

The value of the PC in absolute terms; that is, the actual address in virtual memory representing the location at which the error occurred.

Traceback information is available at runtime only for modules compiled and linked with the traceback option in effect. The traceback option is in effect by default for both the CC and LINK commands. You may use the CC command qualifier /NODEBUG and the LINK command qualifier /NOTRACEBACK to exclude traceback information. However, traceback information should be excluded only from thoroughly debugged program modules.

1.6. Passing Arguments to the main Function

The `main` function in a VSI C program can accept arguments from the command line from which it was invoked. The syntax for a `main` function is:

```
int main(int argc, char *argv[ ], char *envp[ ])

{…}
```

argc

The number of arguments in the command line that invoked the program.

argv

A pointer to an array of character strings that contain the arguments.

envp

The environment array. It contains process information such as the user name and controlling terminal. It has no bearing on passing command-line arguments. Its primary use in VSI C programs is during `exec` and `getenv` function calls. (For more information, see the *VSI C Run-Time Library Reference Manual for OpenVMS Systems* [https://docs.vmssoftware.com/vsi-c-run-time-library-reference-manual-for-openvms-systems/]).

In the `main` function definition, the parameters are optional. However, you can access only the parameters that you define. You can define the `main` function in any of the following ways:

```
int main()
int main(int argc)
int main(int argc, char *argv[ ])
```

```
int main(int argc, char *argv[ ], char *envp[ ])
```

To pass arguments to the `main` function, you must install the program as a DCL foreign command. When a program is installed and run as a foreign command, the *argc* parameter is always greater than or equal to 1, and *argv*[0]always contains the name of the image file.

The procedure for installing a foreign command involves using a DCL assignment statement to assign the name of the image file to a symbol that is later used to invoke the image. For example:

```
$ ECHO == "$DSK$:COMMARG.EXE"
```

The symbol ECHO is installed as a foreign command that invokes the image in COMMARG.EXE. The definition of ECHO must begin with a dollar sign ($) and include a device name, as shown.

For more information about the procedure for installing a foreign command, see the *VSI OpenVMS DCL Dictionary*.

Example 1.1, "Echo Program Using Command-Line Arguments" shows a program called COMMARG.C, which displays the command-line arguments that were used to invoke it.

Example 1.1. Echo Program Using Command-Line Arguments

```
/*  This program echoes the command-line arguments.           */

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
   int i;
                                    /* argv[0] is program name  */
   printf("program: %s\n",argv[0]);

   for (i = 1;  i < argc;  i++)
      printf("argument %d: %s\n", i, argv[i]);

exit (EXIT_SUCCESS);
}
```

You can compile and link the program using the following DCL command lines:

```
$ CC COMMARG
$ LINK  COMMARG
```

A sample output for *Example 1.1, "Echo Program Using Command-Line Arguments"* follows:

```
$ ECHO  Long  "Day's"  "Journey into Night"
program: db7:[oneill.plays]commarg.exe;1
argument 1: long
argument 2: Day's
argument 3: Journey into Night
```

DCL converts most arguments on the command line to uppercase letters. VSI C internally parses and modifies the altered command line to make VSI C argument access compatible with C programs developed on other systems. All alphabetic arguments in the command line are delimited by spaces or tabs. Arguments with embedded spaces or tabs must be enclosed in quotation marks (" "). Uppercase characters in arguments are converted to lowercase, but arguments within quotation marks are left unchanged.

1.7. 64-bit Addressing Support

OpenVMS 64-bit virtual addressing support makes the 64-bit virtual address space defined by the Alpha and Itanium architectures available to the OpenVMS operating system and its users. It also allows per-process virtual addressing for accessing dynamically mapped data beyond traditional 32-bit limits.

The VSI C compiler supports 64-bit pointers on all hardware platforms where the OpenVMS operating system supports 64-bit pointers; that is, on the Alpha and Itanium processors.

This support is provided through command-line qualifiers and pragma preprocessor directives that control the size of the C pointer because:

- Typical C usage involves many objects accessed through pointers rather than single monolithic arrays or structures.
- Huge declared objects would have an impact on object-module format and the linker.

  Note

  Single objects larger than 2 gigabytes are not fully supported, even with 64-bit virtual addressing in effect.

- Minimal source-code edits are required to exploit the 64-bit space where needed. Because the pragmas affect a region of source code, it is not necessary to modify every declaration.

No changes are required for existing 32-bit applications that do not need to exploit 64-bit addressing.

1.7.1. Qualifiers and Pragmas

The following qualifiers, pragmas, and predefined macro control pointer size:

- /[NO]POINTER_SIZE={LONG | SHORT | 64 |32}
- /[NO]CHECK=[NO]POINTER_SIZE=(option,...)
- `#pragma pointer_size`
- `#pragma required_pointer_size`
- `__INITIAL_POINTER_SIZE` predefined macro

1.7.1.1. The /POINTER_SIZE Qualifier

The /POINTER_SIZE qualifier lets you specify a value of 64 or 32 (or LONG or SHORT) as the default pointer size within the compilation unit. You can compile one set of modules using 32-bit pointers and another set using 64-bit pointers. Take care when these two separate groups of modules call each other.

The default is /NOPOINTER_SIZE, which:

- Disables pointer-size features, such as the ability to use `#pragma pointer_size`
- Directs the compiler to assume that all pointers are 32-bit pointers.

This default represents no change over previous versions of VSI C.

Specifying /POINTER_SIZE with a keyword value (32, 64, SHORT, or LONG) has the following effects:

- Enables processing of `#pragma pointer_size`.
- Sets the initial default pointer size to 32 or 64, as specified.
- Predefines the preprocessor macro `__INITIAL_POINTER_SIZE` to 32 or 64, as specified. If /POINTER_SIZE is omitted from the command line, `__INITIAL_POINTER_SIZE` is 0, which allows you to use `#ifdef __INITIAL_POINTER_SIZE` to test whether or not the compiler supports 64-bit pointers.
- For /POINTER_SIZE=64, the VSI C RTL name mapping table is changed to select the 64-bit versions of `malloc`, `calloc`, and other RTL routines by default.

Use of the /POINTER_SIZE qualifier also influences the processing of VSI C RTL header files:

- For those functions that have both 32-bit and 64-bit implementations, specifying /POINTER_SIZE enables function prototypes to access both functions, regardless of the actual value supplied to the qualifier. The value specified to the qualifier determines the default implementation to call during that compilation unit.
- Functions that require a second interface to be used with 64-bit pointers reside in the same object libraries and shareable images as their 32-bit counterparts. Because no new object libraries or shareable images are introduced, using 64-bit pointers does not require changes to your link command or link options files.

See the *VSI C Run-Time Library Reference Manual for OpenVMS Systems* [https://docs.vmssoftware.com/vsi-c-run-time-library-reference-manual-for-openvms-systems/] for more information on the impact of 64-bit pointer support on VSI C RTL functions.

See *Section 1.3.4, "CC Command Qualifiers"* for more information about /POINTER_SIZE.

1.7.1.2. The __INITIAL_POINTER_SIZE Macro

The `__INITIAL_POINTER_SIZE` preprocessor macro is useful for header-file authors to determine:

- If the compiler supports 64-bit pointers.
- If the application expects to use 64-bit pointers.

Header-file code can then be conditionalized using the following preprocessor directives:

```
#if defined (__INITIAL_POINTER_SIZE) /* Compiler supports 64-bit pointers
 */
#if __INITIAL_POINTER_SIZE > 0   /* Application uses 64-bit pointers */
#if __INITIAL_POINTER_SIZE == 32 /* Application uses some 64-bit
                                    pointers, but default RTL
                                    routines are 32-bit.*/

#if __INITIAL_POINTER_SIZE == 64 /* Application uses 64-bit
                                    pointers and default RTL
                                    routines are 64-bit. */
```

1.7.1.3. The /CHECK=POINTER_SIZE Qualifier

Use the /CHECK=POINTER_SIZE qualifier to generate code that checks 64-bit pointer values at runtime to make sure they can fit in a 32-bit pointer. If such a value cannot be represented by a 32-bit pointer, the run-time code signals a range error (SS$_RANGEERR).

Be aware that the compiler generates the same kinds of warning messages for pointer-size mismatches whether or not this qualifier is specified. The run-time checks can detect problems that cannot be detected at compile time, and can help determine whether or not certain warnings are safe to suppress.

See *Section 1.3.4, "CC Command Qualifiers"* for more information about /CHECK=POINTER_SIZE, including defaults and an example.

1.7.1.4. Pragmas

The `#pragma pointer_size` and `#pragma required_pointer_size` preprocessor directives can be used to change the pointer size currently in effect within a compilation unit. You can default pointers to 32-bits and then declare specific pointers within the module as 64-bits. In this case, you also need to specifically call the `_malloc64` form of `malloc` to obtain memory from the 64-bit memory area.

These pragmas have the following format:

```
#pragma pointer_size keyword

#pragma required_pointer_size keyword
```

The *keyword* is one of the following:

{`short`|32}	32-bit pointer
{`long`|64}	64-bit pointer
`save`	Saves the current pointer size
`restore`	Restores the current pointer size to its last saved state

The `#pragma pointer_size` and `#pragma required_pointer_size` directives work essentially the same way, except that `#pragma required_pointer_size` always takes effect regardless of command-line qualifiers, while `#pragma pointer_size` is only in effect when the /POINTER_SIZE command-line qualifier is used.

The `#pragma pointer_size` behavior allows a program to be built using 64-bit features as purely as a 32-bit program, just by changing the command-line qualifier.

The `#pragma required_pointer_size` is intended for use in header files where interfaces to system data structures must use a specific pointer size regardless of how the program is compiled.

See Sections *Section 5.4.19, "#pragma pointer_size Directive "* and *Section 5.4.20, "#pragma required_pointer_size Directive "* for more information on the pointer-size pragmas.

1.7.2. Determining Pointer Size

The pointer-size qualifiers and pragmas affect only a limited number of constructs in the C language itself. At places where the syntax creates a pointer type, the pointer-size context determines the size of that type. Pointer-size context is defined by the most recent pragma (or command-line qualifier) affecting pointer size.

Here are examples of places in the syntax where a pointer type is created:

- The * in a declaration or cast:

```
int **p;  // Declaration
ip = (int **)i;  // Cast
```

- The outer (leftmost) brackets [] in a formal parameter imply a *:

```
void foo(int ia[10][20]) {}

// Means the following:

void foo(int (*ia)[20]) {}
```

- A function declarator as a formal parameter imply a *:

```
void foo (int func()):

// Means the following:

void foo (int (*)() func);
```

- Any formal parameter of array or function type implies a *, even when bound in a `typedef`:

```
typedef int a_type[10];

void foo (a_type ia);

// Means the following:

void foo (int *ia);
```

Note that a `typedef` binds the meaning of pointer syntax while a macro does not. Even though both constructs can contain a * used in a declaration, the * in the macro definition is not affected by any pointer-size controls until the point at which the macro is expanded. For example:

```
#pragma pointer_size 64
typedef int * j_ptr;    // * is 64-bit
#define J_PTR int *     // * is not analyzed

#pragma pointer_size 32
j_ptr j;   // j is a 64-bit pointer.
J_PTR J;   // J is a 32-bit pointer.
```

1.7.2.1. Special Cases

The following special cases are not affected by pointer-size context:

- Formal parameters to `main` are always treated as if they were in a `#pragma pointer_size system_default` context, which is 32-bit pointers for OpenVMS systems.

  For example, regardless of the `#pragma pointer_size 64` directive, argv[0] is a 32-bit pointer:

```
#pragma pointer_size 64

main(int argc, char **argv)
```

```
  { ASSERT(sizeof(argv[0]) == 4); }
```

- A string literal produces a 32-bit pointer when used as an rvalue:

```
#pragma pointer_size 64

ASSERT(sizeof("x" + 0) == 4);
```

- The `&` operator yields a 32-bit pointer unless it is applied to pointer dereference, in which case it is the size of the dereferenced pointer type:

```
sizeof(&foo) == 32

sizeof(&s ->next) == sizeof(s)
```

- An rvalue cast to a 32-bit pointer type does not modify the high-order 32 bits of a 64-bit operand. `sizeof` yields 4 bytes, but the high bits are not lost unless a 4-byte assignment occurs:

```
#pragma pointer_size 64
typedef int * ip64;

#pragma pointer_size 32
typedef int * ip32;

ip64 a,b;
ip32 c;

a = (ip32)b;   // No high-order bits are lost
c = (ip32)b;   // High-order bits are lost
```

1.7.2.2. Mixing Pointer Sizes

An application can use both 32-bit and 64-bit addresses. The following semantics apply when mixing pointers:

- Assignments (including arguments) silently promote a 32-bit pointer rvalue to 64 bits if other type rules are met. Promotion means sign extension.
- A warning is issued for an assignment of a 64-bit rvalue to a 32-bit lvalue (without an explicit cast).
- For purposes of type compatibility, a different size pointer is a different type (for example, when matching a prototype to a definition, or other contexts involving redeclaration).
- The debugger knows the difference between pointers of different sizes.

1.7.3. Header File Considerations

The following general header-file considerations should be kept in mind:

- Header files usually define interfaces with types that must match the layout used in library modules.
- Header files often do not bind "top-level" pointer types. Consider, for example:

```
fprintf(FILE *, const char *, ...);
```

  A "FILE * fp;" in a declaration in a different area of source code might be a different size.

- All pointer parameters occupy 64 bits in the OpenVMS Alpha and I64 calling sequence, so a top-level mismatch of this kind is all right if the called function does not lose the high bits internally.
- Routines dealing with pointers to pointers (or data structures containing pointers) cannot be enabled to work simply by passing them both 32-bit and 64-bit pointers. You need to have separate 32-bit and 64-bit variants of the routine.
- The VSI C RTL header files and the compiler cooperatively provide dual implementations of functions that need to know the pointer size used by the caller. They have different names. The compiler automatically calls the appropriate name within the pointer-size context if the source code calls the simple name. For example, a call to `malloc` becomes:
  - `_malloc64` if /POINTER_SIZE=64.
  - `_malloc32` if /POINTER_SIZE=32.
  - `malloc` if /POINTER_SIZE is omitted.

  If /POINTER_SIZE is specified alone or with a value, `_malloc64` or `_malloc32` can be called explicitly. If /POINTER_SIZE is not specified, the program is compiled to be unaware of 64-bit pointers, and so the declarations of these alternate variants are suppressed.

Be aware that pointer-size controls are not unique in the way they affect header files; other features that affect data layout have similar impact. For example, most header files should be compiled with 32-bit pointers regardless of pointer-size context. Also, most system header files (on OpenVMS Alpha and IA-64 systems) must be compiled with `member_alignment` regardless of user pragmas or qualifiers.

To address this issue more generally, the `pragma environment` directive can be used to save context and set header defaults at the beginning of each header file, and then to restore context at the end. See *Section 5.4.4, "#pragma environment Directive"* for a description of `pragma environment`.

For header files that have not yet been upgraded to use #pragma environment, the /POINTER_SIZE=64 qualifier can be difficult to use effectively. For such header files that are not 64-bit aware, the compiler automatically applies user-defined prologue and epilogue files before and after the text of the included header file. See *Section 1.7.4, "Prologue/Epilogue Files"* for more information on prologue/epilogue files.

1.7.4. Prologue/Epilogue Files

VSI C automatically processes user-supplied prologue and epilogue header files. This feature is an aid to using header files that are not 64-bit aware within an application that is built to exploit 64-bit addressing.

1.7.4.1. Rationale

VSI C header files typically contain a section at the top that:

1. Saves the current state of the `member_alignment`, `extern_model`, `extern_prefix`, and `message` pragmas.
2. Sets these pragmas to the default values for the system.

A section at the end of the header file then restores these pragmas to their previously-saved state.

Mixed pointer sizes introduce another kind of state that typically needs to be saved, set, and restored in header files that define fixed 32-bit interfaces to libraries and data structures.

The `#pragma environment` preprocessor directive allows headers to control all compiler states (message suppression, `extern_model`, `member_alignment`, and `pointer_size`) with one directive.

However, for header files that have not yet been upgraded to use `#pragma environment`, the /POINTER_SIZE=64 qualifier can be difficult to use effectively. In this case, the automatic mechanism to include prologue/epilogue files allows you to protect all of the header files within a single directory (or modules within a single text library). You do this by copying two short files into each directory or library that needs it, without having to edit each header file or library module separately.

In time, you should modify header files to either exploit 64-bit addressing (like the VSI C RTL), or to protect themselves with `#pragma environment`. Prologue/epilogue processing can ease this transition.

1.7.4.2. Using Prologue/Epilogue Files

Prologue/epilogue file are processed in the following way:

1. When the compiler encounters an `#include` preprocessing directive, it determines the location of the file or text library module to be included. It then checks to see if one or both of the two following specially named files or modules exist in the same location as the included file:

   ```
   __DECC_INCLUDE_PROLOGUE.H
   __DECC_INCLUDE_EPILOGUE.H
   ```

   The location is the OpenVMS directory containing the included file or the text library file containing the included module. (In the case of a text library, the .h is stripped off.)

   The directory is the result of using the $PARSE/$SEARCH system services with concealed device name logicals translated. Therefore, if an included file is found through a concealed device logical that hides a search list, the check for prologue/epilogue files is still specific to the individual directories making up the search list.

2. If the prologue and epilogue files do exist in the same location as the included file, then the content of each is read into memory.

3. The text of the prologue file is processed *just before* the text of the file specified by the `#include`.

4. The text of the epilogue file is processed *just after* the text of the file specified by the `#include`.

5. Subsequent `#includes` that refer to files from the same location use the saved text from any prologue/epilogue file found there.

The prologue/epilogue files are otherwise treated as if they had been included explicitly: `#line` directives are generated for them if /PREPROCESS_ONLY output is produced, and they appear as dependencies if /MMS_DEPENDENCY output is produced.

To take advantage of prologue/epilogue processing for included header files, you need to create two files, `__DECC_INCLUDE_PROLOGUE.H` and `__DECC_INCLUDE_EPILOGUE.H`, in the same directory as the included file.

Suggested content for a prologue file is:

```
__DECC_INCLUDE_PROLOGUE.H:

#ifdef __PRAGMA_ENVIRONMENT
```

```
#pragma environment save
#pragma environment header_defaults
#else
#error "__DECC_INCLUDE_PROLOGUE.H: This compiler does not support
pragma environment"
#endif
```

Suggested content for an epilogue file is:

```
__DECC_INCLUDE_EPILOGUE.H:

#ifdef __PRAGMA_ENVIRONMENT
#pragma __environment restore
#else
#error "__DECC_INCLUDE_EPILOGUE.H: This compiler does not support
pragma environment"
#endif
```

1.7.5. Avoiding Problems

Consider the following suggestions to avoid problems related to pointer size:

- Write code to work with either 32-bit or 64-bit pointers by using only the /POINTER_SIZE qualifier.
- Do bit manipulation on `unsigned int` and `unsigned __int64`, and carefully cast pointers to and from them.
- Heed compile-time warnings, using casts only where you are sure that pointers are not truncated.
- Enable the optional compile-time warning (/WARN=ENABLE=MAYHIDELOSS).
- Do thorough testing when compiling with /CHECK=POINTER_SIZE.

1.7.6. Examples

The following examples illustrate the use and misuse of 64-bit pointers.

Example 1.2. Watch Out for Pointers to Pointers (**)

```
/* CC/NAME=AS_IS/POINTER_SIZE=64 */

#include <stdio.h>

#pragma pointer_size 64
char  *C[2]    = {"AB", "CD"};  /* sizeof(C) = 16           */
char **CPTRPTR = C;
char **CPTR;

#pragma pointer_size 32
char  *c[2]    = {"ab", "cd"};  /* sizeof(C) = 8            */
char **cptrptr = c;
char **cptr;

int main (void)
{
    CPTR    =  cptr;           /* No problem.              */
```

```
    cptr    =  CPTR;           /* %CC-W-MAYLOSEDATA         */

    CPTRPTR = cptrptr;         /* %CC-W-PTRMISMATCH         */
    cptrptr = CPTRPTR;         /* MAYLOSEDATA & PTRMISMATCH */
    puts(cptrptr[0]);          /* ab                        */
    puts(cptrptr[1]);          /* cd                        */
    puts(CPTRPTR[0]);          /* Bad address passed.       */
    puts(CPTRPTR[1]);          /* Fetch off end of c.       */
}
```

Compiling *Example 1.2, "Watch Out for Pointers to Pointers (**)"* produces:

```
$ cc example1/name=as_is/pointer_size
    cptr    =  CPTR;           /* %CC-W-MAYLOSEDATA         */
....^
%CC-W-MAYLOSEDATA, In this statement, "CPTR" has a larger
data size than "short pointer to char".  Assignment may
result in data loss.)

    CPTRPTR = cptrptr;         /* %CC-W-PTRMISMATCH         */
....^
%CC-W-PTRMISMATCH, In this statement, the referenced type
of the pointer value "cptrptr" is "short pointer to char",
which is not compatible with "long pointer to char".

    cptrptr = CPTRPTR;         /* MAYLOSEDATA & PTRMISMATCH */
....^
%CC-W-MAYLOSEDATA, In this statement, "CPTRPTR" has a
larger data size than "short pointer to short pointer
to char".  Assignment may result in data loss.)

    cptrptr = CPTRPTR;         /* MAYLOSEDATA & PTRMISMATCH */
....^
%CC-W-PTRMISMATCH, In this statement, the referenced type
of the pointer value "CPTRPTR" is "long pointer to char",
which is not compatible with "short pointer to char".
```

Example 1.3. Trivial 64-Bit Exploitation

```
#include <stdio.h>
#include <stdlib.h>
__int64 limit, count;
size_t bytes;
char  *cp, *prevcp;

int main(int argc, char **argv)
{
    sscanf(argv[1], "%d",  &bytes);
    sscanf(argv[2], "%Ld", &limit);
    printf("bytes %d, limit %Ld, tot %Ld\n",
           bytes,     limit, bytes * limit);
    for (count=0; count < limit; count++)  {
        if (!(cp = malloc(bytes)))  {
           printf("Max %Ld bytes.\n", bytes * (count + 1));
           break;
    } else if (!prevcp)
        printf("First addr %Lx.\n", cp);
    }
```

```
    prevcp = cp;
    printf("Last addr %Lx.\n", prevcp);
}
```

Compiling, linking, and running *Example 1.3, "Trivial 64-Bit Exploitation"* produces:

```
$ cc example2
$ link example2

$ example2:==$sys$login:[.john]example2 ! << set up a symbol
$ example2 65536 1234567890123456
bytes 65536, limit 1234567890123456, tot 7121664952292605952

First addr 610b0.
First addr 730b0.
First addr 850b0.
First addr 970b0.
First addr a90b0.

   .
   .
   .

First addr f1c30b0.
First addr f1d50b0.
First addr f1e70b0.
First addr f1f90b0.
First addr f20b0b0.
Max 225378304 bytes.
Last addr 0.


$
$ cc/pointer_size=64 example2
$ link example2

$ example2 65536 1234567890123456
bytes 65536, limit 1234567890123456, tot
7121664952292605952
First addr 1c0010010.
Max 42532864 bytes.
Last addr 1c2d8e010.
```

Example 1.4. Preceding Example No Longer Trivial

```
#include <stdio.h>
#include <stdlib.h>
__int64 limit, count;
size_t bytes;
char  *cp, *prevcp;

static void do_args(char **args)
{
    sscanf(argv[1], "%d",  &bytes);
    sscanf(argv[2], "%Ld", &limit);
    printf("bytes %d, limit %Ld, tot %Ld\n",
           bytes,     limit, bytes * limit);
}
```

```
int main(int argc, char **argv)
{
    do_args(argv);
    for (count=0; count < limit; count++)  {
        if (!(cp = malloc(bytes)))  {
           printf("Max %Ld bytes.\n", bytes * (count + 1));
           break;
    } else if (!prevcp) {
        printf("First addr %Lx.\n", cp);
    }
    prevcp = cp;
    printf("Last addr %Lx.\n", prevcp);
}
```

Compiling *Example 1.4, "Preceding Example No Longer Trivial"* produces:

```
$ cc/pointer_size=64 example3
do_args(argv);
....^
%CC-W-PTRMISMATCH, In this statement, the referenced type
of the pointer value "argv" is "short pointer to char",
which is not compatible with "long pointer to char".
```

Chapter 2. Using OpenVMS Record Management Services

VSI C for OpenVMS systems provides a set of run-time library functions and macros to perform I/O. Some of these functions perform in the same manner as I/O functions found on C implementations running on UNIX systems. Other VSI C functions take full advantage of the functionality of the OpenVMS file-handling system. You can also access the OpenVMS file-handling system from your VSI C program without using the VSI C Run-Time Library (C RTL) functions. In any case, the system that ultimately accesses files on OpenVMS systems is OpenVMS Record Management Services (RMS).

This chapter introduces you to the following RMS topics:

- RMS file organization (*Section 2.1, "RMS File Organization"*)
- Record access modes (*Section 2.2, "Record Access Modes"*)
- RMS record formats (*Section 2.3, "RMS Record Formats"*)
- RMS functions (*Section 2.4, "RMS Functions"*)
- Writing VSI C programs using RMS (*Section 2.5, "Writing VSI C Programs Using RMS"*)
- RMS example program (*Section 2.6, "RMS Example Program"*)

The file-handling capabilities of VSI C fall into two distinct categories:

- The VSI C RTL functions which, with little or no modification, are portable to other C implementations
- The RMS functions, which are not portable to other C implementations, but do provide more methods of file organization and more record access modes

This chapter briefly reviews the basic concepts and facilities of RMS and shows examples of their application in VSI C programming. Because this is an overview, the chapter does not explain all RMS concepts and features. For language-independent information about RMS, see the following manuals in the OpenVMS documentation set:

- *Guide to OpenVMS File Applications*

  This guide contains a general description of the record management services of the OpenVMS operating system, and the file creation and run-time options available.

- *OpenVMS Record Management Services Reference Manual*

  This manual describes the user interface to RMS. It includes introductory information on RMS programming and detailed definitions of all RMS control block structures and macro instructions.

2.1. RMS File Organization

RMS supports three types of file organization:

- Sequential

- Relative
- Indexed

The following sections describe these types of file organization.

The organization of a file determines how a file is stored on the media and, consequently, the possible operations on records. You specify the file's organization when you create the file; it cannot be changed.

However, you can use the File Definition Language Editor (FDL) and the CONVERT utility to define the characteristics of a new file, and then fill the new file with the contents of the old file of a different format. For more information, see the *OpenVMS Utility Routines Manual*.

2.1.1. Sequential File Organization

Sequential files have consecutive records. There are no empty records separating records that contain data. This organization allows the following operations on the file:

- Positioning the file at a particular record, generally by sequentially moving from one record to the next.

  Direct access is also possible, either by key (relative record number) or by the record file address (RFA). However, although allowed for any file organization, access by RFA is limited to files on disk devices, and access by key is limited to disk files that also have fixed-length records. These access modes are unusual because most application programs do not keep track of record positions in sequential files.

- Reading data from any record.
- Writing data by adding records at the end of the file.

Sequential organization is the only kind permitted for magnetic tape files and other nondisk devices.

2.1.2. Relative File Organization

Relative files have records that occupy numbered, fixed-length cells. The records themselves need not have the same length. Cells can be empty or can contain records so the following operations are permitted:

- Positioning the file at a particular record, usually by direct access.

  In direct access, RMS uses the relative record number—the number of a cell—as a key to locate the cell and its record; there is no need to reference other cells. RMS can also access the records sequentially by ignoring empty cells, or RMS can access the file directly with the record file address (RFA). RMS returns the RFA in a parameter block whenever it writes a record, and you can access and use the RFA to locate the appropriate record. You can access any file organization with the RFA.

- Reading a record from any cell.
- Deleting a record from any cell.
- Writing a record into any cell.

Relative file organization is possible only on disk devices.

2.1.3. Indexed File Organization

Indexed files have records that contain, in addition to data and carriage-control information, one or more keys. Keys can be character strings, packed decimal numbers, and 16-bit, 32-bit, or 64-bit signed or unsigned integers. Every record has at least one key, the primary key, whose value in each record cannot be changed. Optionally, each record can have one or more alternate keys, whose key values can be changed.

Unlike relative record numbers used in relative files, key values in indexed files are not necessarily unique. When you create a file, you can specify that a particular key have the same value in different records (these keys are called duplicate keys). Keys are defined for the entire file in terms of their position within a record and their length.

In addition to maintaining its records, RMS builds and maintains indexes for each of the defined keys. As records are written to the file, their key values are inserted in order of ascending value in the appropriate indexes. This organization allows the following operations:

- Positioning the file at a particular record by direct access.

  In direct access reads, you use either a primary or alternate key, plus a specified key value, to locate the record. In direct access writes (given a record that contains key values in the predefined positions), RMS automatically adds the record to the file and adds the primary and alternate key values to the appropriate indexes. You can also access records sequentially, where the sequence is defined by the index for a specified key. Finally, you can access records directly by RFA; RMS returns the RFA in a parameter block whenever it writes a record, and you can access and use the RFA to locate the appropriate record. You can access any file organization with the RFA.

- Reading any record, including sequential reads controlled by a key's index.

- Deleting any record.

- Updating an alternate key's value, if the key's definition permits its value to change.

- Writing records selectively, based on the value of a key and, when allowed in the key's definition, based on duplicate values. If duplicate values are permitted, you can write records containing key values that are present in the key's index. If duplicate values are not permitted, such write operations are rejected.

Indexed organization is possible only on disk devices.

2.2. Record Access Modes

The record access modes are sequential, direct by key, and direct by record file address. The direct access modes are possible only with files that reside on disks.

Unlike a file's organization, the record access mode is not a permanent attribute of the file. During the processing of a file, you can switch from one access mode to any other permitted for that file organization. For example, indexed files are often processed by locating a record directly by key, and then using that key's index to sequentially read all the indexed records in ascending order of their key values; this method is sometimes called the indexed-sequential access method (ISAM).

2.3. RMS Record Formats

Records in RMS files can have the following formats:

- Fixed-length format, where the length of every record is defined at the time of the file's creation. This format is permitted with any file organization.
- Variable-length format, where the maximum length of every record is defined at the time of the file's creation. This format is permitted with any file organization.
- Variable-length format with a fixed-length control area (VFC), where every record is prefixed by a fixed-length field. This format is permitted only with sequential and relative files.
- Stream format, where records are delimited by special characters called *terminators*. Terminators are part of the record they delimit. The three types of stream formatting are as follows:
  - Stream, where records can be delimited with a form feed, vertical tab, new-line character, or carriage-return/new-line character.
  - Stream_cr, where records are delimited with the carriage-return character.
  - Stream_lf, where records are delimited with the line-feed character. This format variation is the default format when you create files using the Standard I/O functions.

2.4. RMS Functions

RMS provides a number of functions that create and manipulate files. These functions use RMS data structures to define the characteristics of a file and its records. The data structures are used as indirect arguments to the function call.

The RMS data structures are grouped into four main categories, as follows:

- File Access Block (FAB) – Defines the file's characteristics, such as file organization and record format.
- Record Access Block (RAB) – Defines the way in which records are processed, such as the record access mode.
- Extended Attribute Block (XAB) – Various kinds of extended attribute blocks contain additional file characteristics, such as the definition of keys in an indexed file. Extended attribute blocks are optional.
- Name Block (NAM) – Defines all or part of a file specification to be used when an incomplete file specification is given in an OPEN or CREATE operation. Name blocks are optional.

RMS uses these data structures to perform file and record operations. *Table 2.1, "Common RMS Run-Time Processing Functions"* lists some of the common functions.

Table 2.1. Common RMS Run-Time Processing Functions

Category	Function	Description
File Processing	sys$create	Creates and opens a new file of any organization.
	sys$open	Opens an existing file and initiates file processing.
	sys$close	Terminates file processing and closes the file.

Category	Function	Description
	sys$erase	Deletes a file.
Record Processing	sys$connect	Associates a file access block with a record access block to establish a record access stream; a call to this function is required before any other record-processing function can be used.
	sys$get	Retrieves a record from a file.
	sys$put	Writes a new record to a file.
	sys$update	Rewrites an existing record to a file.
	sys$delete	Deletes a record from a file.
	sys$rewind	Positions the record pointer to the first record in the file.
	sys$disconnect	Disconnects a record access stream.

All RMS functions are directly accessible from VSI C programs. The syntax for any RMS function has the following form:

```
int  sys$name(struct rms_structure *pointer);
```

In this syntax, *name* is the name of the RMS function (such as OPEN or CREATE); *rms_structure* is the name of the structure being used by the function.

The file-processing functions require a pointer to a file access block as an argument; the record-processing functions require a pointer to a record access block as an argument. Since sys$create is a file-processing function, its syntax is as follows:

```
int  sys$create(struct FAB *fab);
```

These syntax descriptions do not show all the options available when you invoke an RMS function. For a complete description of the RMS calling sequence, see the *OpenVMS Record Management Services Reference Manual*.

Finally, all the RMS functions return an integer status value. The format of RMS status values follows the standard format described in *Chapter 3, "Using VSI C in the Common Language Environment"*. Since RMS functions return a 32-bit integer, you do not need to declare the type of an RMS function return before you use it.

2.5. Writing VSI C Programs Using RMS

The VSI C Run-Time Library (C RTL) supplies a number of header files that describe the RMS data structures and status codes. *Table 2.2, "VSI C RMS Header Files"* describes these header files.

Table 2.2. VSI C RMS Header Files

Header File	Structure Tag(s)	Description
`<fab.h>`	FAB	Defines the file access block structure.
`<rab.h>`	RAB	Defines the record access block structure.
`<nam.h>`	NAM	Defines the name block structure.

Header File	Structure Tag(s)	Description
<xab.h>	XAB	Defines all the extended attribute block structures.
<rmsdef.h>	–	Defines the completion status codes that RMS returns after every file- or record-processing operation.
<rms.h>	all tags	Includes all the previous header files.

Most VSI C programmers include the <rms.h> header file, which includes all the other header files.

These header files define all the data structures as structure tag names. However, they perform no allocation or initialization of the structures; these header files describe only a template for the structures. To use the structures, you must create storage for them and initialize all the structure members as required by RMS. Note that these include files are part of VSI C for OpenVMS systems. RMS is part of the OpenVMS environment and may contain other included header files not described here.

To assist in the initialization process, the C RTL provides initialized RMS data structure variables. You can copy these variables to your uninitialized structure definitions with a structure assignment. You can choose to take the default values for each of the structure members, or you can tailor the contents of the structures to fit your requirements. In either case, you must use the structure types to allocate storage for the structure and to define the members of the structure.

The initialized variables supply the RMS default values for each member in the structure; they specify none of the optional parameters. To determine what default values are supplied by the initialized variables, see the *VSI OpenVMS Record Management Services Reference Manual*.

Table 2.3, "RMS Data Structures" lists the initialized RMS data structure variables and the structures that they initialize.

Table 2.3. RMS Data Structures

Variable	Structure Type	Initialize Structure
cc$rms_fab	struct FAB	File access block
cc$rms_rab	struct RAB	Record access block
cc$rms_nam	struct NAM	Name block
cc$rms_xaball	struct XABALL	Allocation extended attribute block
cc$rms_xabdat	struct XABDAT	Date and time extended attribute block
cc$rms_xabfhc	struct XABFHC	File header characteristics extended attribute block
cc$rms_xabkey	struct XABKEY	Indexed file key extended attribute block
cc$rms_xabpro	struct XABPRO	Protection extended attribute block
cc$rms_xabrdt	struct XABRDT	Revision date and time extended attribute block
cc$rms_xabsum	struct XABSUM	Summary extended attribute block
cc$rms_xabtrm	struct XABTRM	Terminal extended attribute block

The declarations of these structures are contained in the appropriate header file.

The names of the structure members conform to the following RMS naming convention:

```
typ$s_fld
```

The identifier *typ* is the abbreviation for the structure, the letter *s* is the size of the member (such as l for longword or b for byte), and the identifier *fld* is the member name, such as sts for the completion status code. The dollar sign ($) is a character used in OpenVMS system logical names. See the *OpenVMS Record Management Services Reference Manual* for a description of the members in each structure.

2.5.1. Initializing File Access Blocks

The file access block defines the attributes of the file. To initialize a file access block, assign the values in the initialized data structure cc$rms_fab to the address of the file access block defined in your program. Consider the following example:

```
/*  This example shows how to initialize a file access block.   */

#include <rms.h>               /*  Declare all RMS data structs */

struct  FAB   fblock;          /*  Define a file access block   */


main()
{
   fblock = cc$rms_fab;        /*  Initialize the structure     */
      .
      .
      .
}
```

Any of these RMS structures may be dynamically allocated. For example, another way to allocate a file access block is as follows:

```
/* This program shows how to dynamically allocate RMS structures. */

#include <rms.h>                 /*  Declare all RMS data structs */


main()
{
                                 /*  Allocate dynamic storage     */
   struct  FAB     *fptr = malloc(sizeof (struct FAB));
   *fptr = cc$rms_fab;           /*  Initialize the structure     */
      .
      .
      .
}
```

To change the default values supplied by a data structure variable, you must reinitialize the members of the structure individually. You initialize a member by giving the offset of the member and assigning a value to it. Consider the following example:

```
fblock.fab$l_xab = &primary_key;
```

This statement assigns the address of the extended attribute block named `primary_key` to the `fab $l_xab` member of the file access block named `fblock`.

2.5.2. Initializing Record Access Blocks

The record access block specifies how records are processed. You initialize a record access block the same way you initialize a file access block. For example:

```
/*  This example shows how to initialize a file access block.  */

#include <rms.h>
struct  FAB  fblock;

struct  RAB  rblock;          /*  Define a record access block */


main()
{
   fblock = cc$rms_fab;       /*  Initialize the structure     */
   rblock = cc$rms_rab;

                              /*  Initialize the FAB member    */
   rblock.rab$l_fab  =  &fblock;
      .
      .
      .
}
```

2.5.3. Initializing Extended Attribute Blocks

There is only one extended attribute block structure (XAB), but there are seven ways to initialize it. The extended attribute blocks define additional file attributes that are not defined elsewhere. For example, the key extended attribute block is used to define the keys of an indexed file.

All extended attribute blocks are chained off a file access block in the following manner:

1. In a file access block, you initialize the fab$l_xab field with the address of the first extended attribute block.

2. You designate the next extended attribute block in the chain in the xab$l_nxt field of any subsequent extended attribute blocks. You chain each subsequent extended attribute block in order by the key of reference (first the primary key, then the first alternate key, then the second alternate key, and so forth).

3. You initialize the xab$l_nxt member of the last extended attribute block in the chain with the value 0 (the default) to indicate the end of the chain.

You go through the same steps to declare extended attribute blocks as you would to declare the other RMS data structures:

1. Define the structures by including the appropriate header file.

2. Assign a specific data structure variable to the structure in your program.

3. Initialize the members of the structure with the desired values.

The following example declares two extended attribute block structures. They are initialized as key extended attribute blocks with the cc$rms_xabkey data structure variable. The xab$l_nxt member of the primary key is initialized with the address of the alternate_key extended attribute block.

```
/*  This example shows how to initialize the extended          *
 *  attribute block.                                           */

#include <rms.h>
```

```
struct  XABKEY  primary_key,alternate_key;


main()
{
   primary_key           = cc$rms_xabkey;
   alternate_key         = cc$rms_xabkey;
   primary_key.xab$l_nxt = &alternate_key;
      .
      .
      .
}
```

2.5.4. Initializing Name Blocks

The name block contains default file name values, such as the directory or device specification, file name, or file type. If you do not specify one of the parts of the file specification when you open the file, RMS uses the values in the name block to complete the file specification and places the complete file specification in an array.

You create and initialize name blocks in the same manner used to initialize the other RMS data structures. Consider the following example:

```
/*  This example shows how to initialize a name block.         */

#include <rms.h>

struct  NAM  nam;
struct  FAB  fab;


main()
{
   fab = cc$rms_fab;
   nam = cc$rms_nam;


                              /*  Define an array for the      *
                               *   expanded file specification */
   char expanded_name[NAM$C_MAXRSS];

                              /*  Initialize the appropriate   *
                               *   members                     */
   fab.fab$l_nam = &nam;
   nam.nam$l_esa = &expanded_name;
   nam.nam$b_ess =  sizeof  expanded_name;
      .
      .
      .
}
```

2.6. RMS Example Program

The example program in this section uses RMS functions to maintain a simple employee file. The file is an indexed file with two keys: social security number and last name. The fields in the record are character strings defined in a structure with the tag record.

The records have the carriage-return attribute. Individual fields in each record are padded with blanks for two reasons. First, because RMS requires that the key fields be a fixed length and occur in a fixed position in each record, key fields must be padded in some way. The example program pads short fields; its use of the space character for padding is arbitrary. Second, the choice of blank padding (as opposed to null padding) allows the file to be printed or typed without conversion. Note that both the position and size of the key are attributes of the file, not of each I/O that gets done.

The program does not perform range or bounds checking. Only the error checking that shows the mapping of VSI C to RMS is performed. Any other errors are considered fatal.

The program is divided into the following sections:

- External data declarations and definitions
- Main program section
- Function to initialize the RMS data structures
- Internal functions to open the file, display HELP information, pad the records, and process fatal errors
- Utility functions
  - ADD
  - DELETE
  - TYPE
  - PRINT
  - UPDATE

To run this program, perform the following steps:

1. Create a source file. The name of the source file in this example is RMSEXP.C. For more information about creating source files, see *Chapter 1, "Developing VSI C Programs"*.
2. Compile the source file with the following command:

   ```
   $ CC  RMSEXP
   ```

   For more information about the compiling process, see *Chapter 1, "Developing VSI C Programs"*.
3. Link the program with the following command:

   ```
   $ LINK RMSEXP
   ```

   For more information about the linking process, see *Chapter 1, "Developing VSI C Programs"*.
4. Because the program expects command-line arguments, it must be defined as a foreign command. You can do this with the following command line:

   ```
   $ RMSEXP :== $device:[directory]RMSEXP
   ```

   The identifier device is the logical or physical name of the device containing your directory; the identifier directory is the name of your directory. The device name must be preceded by the dollar sign ($) to be recognized as a foreign command by the DCL interpreter.

5. Run the program using the following foreign command:

```
$ RMSEXP  filename
```

The complete listing of the sample program follows. The listing is broken into sections and shown in Examples *Example 2.1, "External Data Declarations and Definitions"* through *Example 2.9, "Utility Function: Updating the File"*. Notes on each section are keyed to the numbers in the listing.

Example 2.1, "External Data Declarations and Definitions" shows the external data declarations and definitions.

Example 2.1. External Data Declarations and Definitions

```
/* This segment of RMSEXP.C contains external data        *
 * definitions.                                           */

❶#include <rms.h>
#include <stdio.h>
#include <ssdef.h>
#include <string.h>
#include <stdlib.h>
#include <starlet.h>

❷#define  DEFAULT_FILE_EXT        ".dat"

#define RECORD_SIZE              (sizeof record)
#define SIZE_SSN                 15
#define SIZE_LNAME               25
#define SIZE_FNAME               25
#define SIZE_COMMENTS            15
#define KEY_SIZE                 \
(SIZE_SSN > SIZE_LNAME ? SIZE_SSN: SIZE_LNAME)

❸struct  FAB fab;
struct  RAB rab;
struct  XABKEY primary_key,alternate_key;

❹struct
   {
      char     ssn[SIZE_SSN], last_name[SIZE_LNAME];
      char     first_name[SIZE_FNAME],
               comments[SIZE_COMMENTS];
   }  record;

❺char  response[BUFSIZ],*filename;

❻int  rms_status;

void open_file(void);
void type_options(void);
void pad_record(void);
void error_exit(char *);
void add_employee(void);
void delete_employee(void);
void type_employees(void);
void print_employees(void);
void update_employee(void);
void initialize(char *);
```

Key to *Example 2.1, "External Data Declarations and Definitions"*:

❶ The <rms.h> header file defines the RMS data structures. The <stdio.h> header file contains the Standard I/O definitions. The <ssdef.h> header file contains the system services definitions.

❷ Preprocessor variables and macros are defined. A default file extension .DAT is defined.

The sizes of the fields in the record are also defined. Some (such as the social security number field) are given a constant length. Others (such as the record size) are defined as macros; the size of the field is determined with the `sizeof` operator. VSI C evaluates constant expressions, such as `KEY_SIZE,` at compile time. No special code is necessary to calculate this value.

❸ Static storage for the RMS data structures is declared. The file access block, record access block, and extended attribute block types are defined by the <rms.h> header file. One extended attribute block is defined for the primary key and one is defined for the alternate key.

❹ The records in the file are defined using a structure with four fields of character arrays.

❺ The `BUFSIZ` constant is used to define the size of the array that will be used to buffer input from the terminal. The file-name variable is defined as a pointer to `char`.

❻ The variable `rms_status` is used to receive RMS return status information. After each function call, RMS returns status information as an integer. This return status is used to check for specific errors, end-of-file, or successful program execution.

The `main` function, shown in *Example 2.2, "Main Program Section"*, controls the general flow of the program.

Example 2.2. Main Program Section

```
/* This segment of RMSEXP.C contains the main function    *
 * and controls the flow of the program.                  */

❶main(int argc, char **argv)
{
❷   if (argc < 1 || argc > 2)
      printf("RMSEXP - incorrect number of arguments");
   else
      {

         printf("RMSEXP - Personnel Database \
                 Manipulation Example\n");

❸         filename = (argc == 2 ? *++argv : "personnel.dat");
❹         initialize(filename);
❺         open_file();


         for(;;)
            {
❻              printf("\nEnter option (A,D,P,T,U) or \
? for help :");

               gets(response);
               if (feof(stdin))
                  break;
               printf("\n\n");
```

```
❼                 switch(response[0])
                    {
                       case 'a': case 'A':  add_employee();
                                            break;

                       case 'd': case 'D':  delete_employee();
                                            break;

                       case 'p': case 'P':  print_employees();
                                            break;

                       case 't': case 'T':  type_employees();
                                            break;

                       case 'u': case 'U':  update_employee();
                                            break;

                       default:             printf("RMSEXP - \
                                                    Unknown Operation.\n");

                       case '?': case '\0':
                                            type_options();
                    }
              }

❽          rms_status = sys$close(&fab);

❾          if (rms_status != RMS$_NORMAL)
                       error_exit("$CLOSE");
       }
}
```

Key to *Example 2.2, "Main Program Section"*:

❶ The `main` function is entered with two parameters. The first is the number of arguments used to call the program; the second is a pointer to the first argument (file name).

❷ This statement checks that you used the correct number of arguments when invoking the program.

❸ If a file name is included in the command line to execute the program, that file name is used. If a file extension is not given, .DAT is the file extension. If no file name is specified, then the file name is PERSONNEL.DAT.

❹ The file access block, record access block, and extended attribute blocks are initialized.

❺ The file is opened using the RMS sys$open function.

❻ The program displays a menu and checks for end-of-file (the character Ctrl/Z).

❼ A `switch` statement and a set of `case` statements control the function to be called, which is determined by the response from the terminal.

❽ The program ends when Ctrl/Z is entered in response to the menu. At that time, the RMS sys$close function closes the employee file.

❾ The `rms_status` variable is checked for a return status of RMS$_NORMAL. If the file is not closed successfully, then the error-handling function terminates the program.

Example 2.3, "Function Initializing RMS Data Structures" shows the function that initializes the RMS data structures. See the RMS documentation for more information about the file access block, record access block, and extended attribute block structure members.

Example 2.3. Function Initializing RMS Data Structures

```
/* This segment of RMSEXP.C contains the function that    *
 * initializes the RMS data structures.                   */

void initialize(char *fn)
{

❶   fab =  cc$rms_fab;               /* Initialize FAB     */
   fab.fab$b_bks =  4;
   fab.fab$l_dna =  DEFAULT_FILE_EXT;
   fab.fab$b_dns =  sizeof DEFAULT_FILE_EXT -1;
   fab.fab$b_fac =  FAB$M_DEL | FAB$M_GET |
                    FAB$M_PUT | FAB$M_UPD;
   fab.fab$l_fna =  fn;
   fab.fab$b_fns =  strlen(fn);
❷   fab.fab$l_fop =  FAB$M_CIF;
   fab.fab$w_mrs =  RECORD_SIZE;
   fab.fab$b_org =  FAB$C_IDX;
❸   fab.fab$b_rat =  FAB$M_CR;
   fab.fab$b_rfm =  FAB$C_FIX;
   fab.fab$b_shr =  FAB$M_NIL;
   fab.fab$l_xab =  &primary_key;


❹   rab =  cc$rms_rab;              /* Initialize RAB      */

   rab.rab$l_fab =  &fab;

❺   primary_key =  cc$rms_xabkey;   /* Initialize Primary  *
                                    *  Key XAB            */
   primary_key.xab$b_dtp = XAB$C_STG;
   primary_key.xab$b_flg = 0;
❻   primary_key.xab$w_pos0 =  (char *) &record.ssn -
                           (char *) &record;
   primary_key.xab$b_ref =  0;
   primary_key.xab$b_siz0 = SIZE_SSN;
   primary_key.xab$l_nxt =  &alternate_key;
   primary_key.xab$l_knm =  "Employee Social Security \
Number       ";

❼   alternate_key = cc$rms_xabkey; /* Initialize Alternate *
                                   *  Key XAB             */
   alternate_key.xab$b_dtp = XAB$C_STG;
❽   alternate_key.xab$b_flg =  XAB$M_DUP | XAB$M_CHG;
   alternate_key.xab$w_pos0 =  (char *) &record.last_name -
                               (char *) &record;
   alternate_key.xab$b_ref =  1;
   alternate_key.xab$b_siz0 =  SIZE_LNAME;
❾   alternate_key.xab$l_knm =   "Employee Last Name \
             ";
}
```

Key to *Example 2.3, "Function Initializing RMS Data Structures"*:

❶ The data structure variable cc$rms_fab initializes the file access block with default values. Some members have no default values; they must be initialized. Such members include the file-name string address and size. Other members can be initialized to override the default values.

❷ This statement initializes the file-processing options member with the create-if option. A file is created if one does not exist.

❸ This statement initializes the record attributes member with the carriage-return control attribute. Records are terminated with a carriage return/line feed when they are printed on the printer or displayed at the terminal.

❹ The data structure variable cc$rms_rab initializes the record access block with the default values. In this case, the only member that must be initialized is the rab$l_fab member, which associates a file access block with a record access block.

❺ The data structure variable cc$rms_xabkey initializes an extended attribute block for one key of an indexed file.

❻ The position of the key is specified by subtracting the offset of the member from the base of the structure.

❼ A separate extended attribute block is initialized for the alternate key.

❽ This statement specifies that more than one alternate key can contain the same value (XAB$M_DUP), and that the value of the alternate key can be changed (XAB$M_CHG).

Note

RMS constants shown here are in the form xxx$M_yyy (for example, RAB$M_FIX) or xxx$C_yyy (for example, RAB$C_FIX). The OpenVMS RMS documentation cites the constants in the form xxx$V_yyy (for example, rab$v_fix), the difference being:

- The $M type constant signifies a bit mask, and should be OR'ed to an existing value.
- The $V type constant represents the bit position of a constant, and a shift operation is necessary for setting the appropriate bit.

Using a $V type constant the same way as a $M type constant is a common problem.

---

❾ The key-name member is padded with blanks because it is a fixed-length, 32-character field.

Example 2.4, "Internal Functions" shows the internal functions for the program.

Example 2.4. Internal Functions

```
/* This segment of RMSEXP.C contains the functions that   *
 * control the data manipulation of the program.          */

void open_file(void)
{
❶   rms_status =  sys$create(&fab);
   if (rms_status != RMS$_NORMAL &&
       rms_status != RMS$_CREATED)
```

```
      error_exit("$OPEN");

   if (rms_status == RMS$_CREATED)
      printf("[Created new data file.]\n");

❷   rms_status =  sys$connect(&rab);
   if (rms_status != RMS$_NORMAL)
      error_exit("$CONNECT");
}


❸void type_options(void)
{
   printf("Enter one of the following:\n\n");
   printf("A     Add an employee.\n");
   printf("D     Delete an employee specified by SSN.\n");
   printf("P     Print employee(s) by ascending SSN on \
line printer.\n");


   printf("T     Type employee(s) by ascending last name \
on terminal.\n");
   printf("U     Update employee specified by SSN.\n\n");
   printf("?     Type this text.\n");
   printf("^Z    Exit this program.\n\n");
}


❹void pad_record(void)
{
   int      i;

   for(i = strlen(record.ssn); i < SIZE_SSN; i++)
      record.ssn[i] = ' ';
   for(i = strlen(record.last_name); i < SIZE_LNAME; i++)
      record.last_name[i] = ' ';
   for(i = strlen(record.first_name); i < SIZE_FNAME; i++)
      record.first_name[i] = ' ';
   for(i = strlen(record.comments);i < SIZE_COMMENTS; i++)
      record.comments[i] = ' ';
}

/* This subroutine is the fatal error-handling routine.   */

❺void error_exit(char *operation)
{
   printf("RMSEXP - file %s failed (%s)\n",
           operation, filename);
   exit(rms_status);
}
```

Key to *Example 2.4, "Internal Functions"*:

❶ The `open_file` function uses the RMS sys$create function to create the file, giving the address of the file access block as an argument. The function returns status information to the `rms_status` variable.

❷ The RMS sys$connect function associates the record access block with the file access block.

❸ The `type_options` function, called from the `main` function, prints help information. Once the help information is displayed, control returns to the `main` function, which processes the response that is typed at the terminal.

❹ For each field in the record, the `pad_record` function fills the remaining bytes in the field with blanks.

❺ This function handles fatal errors. It prints the function that caused the error, returns an OpenVMS error code (if appropriate), and exits the program.

Example 2.5, "Utility Function: Adding Records" shows the function that adds a record to the file. This function is called when '`a`' or '`A`' is entered in response to the menu.

Example 2.5. Utility Function: Adding Records

```
/* This segment of RMSEXP.C contains the function that    *
 * adds a record to the file.                             */

void add_employee(void)
{
❶   do
      {
         printf("(ADD)   Enter Social Security Number:");

         gets(response);

      }
   while(strlen(response) == 0);

   strncpy(record.ssn,response,SIZE_SSN);

   do
      {
         printf("(ADD)    Enter Last Name:");

         gets(response);
      }
   while(strlen(response) == 0);

   strncpy(record.last_name,response,SIZE_LNAME);

   do
      {
         printf("(ADD)    Enter First Name:");

         gets(response);
      }
   while(strlen(response) == 0);

   strncpy(record.first_name,response,SIZE_FNAME);

   do
      {
         printf("(ADD)    Enter Comments:");

         gets(response);
      }
```

```
   while(strlen(response) == 0);

   strncpy(record.comments,response,SIZE_COMMENTS);

❷   pad_record();

❸   rab.rab$b_rac =  RAB$C_KEY;
   rab.rab$l_rbf =  (char *) &record;
   rab.rab$w_rsz =  RECORD_SIZE;

❹   rms_status =  sys$put(&rab);
❺   if (rms_status != RMS$_NORMAL && rms_status !=
                    RMS$_DUP && rms_status != RMS$_OK_DUP)
      error_exit("$PUT");
   else
      if (rms_status == RMS$_NORMAL || rms_status ==
                                                RMS$_OK_DUP)
         printf("[Record added successfully.]\n");
      else
         printf("RMSEXP - Existing employee with same SSN, \
not added.\n");
}
```

Key to *Example 2.5, "Utility Function: Adding Records"*:

❶ A series of `do` loops controls the input of information. For each field in the record, a prompt is displayed. The response is buffered and the field is copied to the structure.

❷ When all fields have been entered, the `pad_record` function pads each field with blanks.

❸ Three members in the record access block are initialized before writing the record. The record access member (rab$b_rac) is initialized for keyed access. The record buffer and size members (rab$l_rbf and rab$w_rsz) are initialized with the address and size of the record to be written.

❹ The RMS sys$put function writes the record to the file.

❺ The `rms_status` variable is checked. If the return status is normal, or if the record has a duplicate key value and duplicates are allowed, the function prints a message stating that the record was added to the file. Any other return value is treated as a fatal error causing `error_exit` to be called.

Example 2.6, "Utility Function: Deleting Records" shows the function that deletes records. This function is called when 'd' or 'D' is entered in response to the menu.

Example 2.6. Utility Function: Deleting Records

```
/* This segment of RMSEXP.C contains the function that    *
 * deletes a record from the file.                        */

void delete_employee(void)
{
   int i;
❶   do
      {
         printf("(DELETE) Enter Social Security Number   ");
         gets(response);
         i =  strlen(response);
      }
```

```
   while(i == 0);

❷   while(i < SIZE_SSN)
      response[i++] = ' ';

❸   rab.rab$b_krf =  0;
   rab.rab$l_kbf =  response;
   rab.rab$b_ksz =  SIZE_SSN;
   rab.rab$b_rac =  RAB$C_KEY;

❹   rms_status =  sys$find(&rab);

❺   if (rms_status != RMS$_NORMAL && rms_status != RMS$_RNF)
      error_exit("$FIND");
   else
      if (rms_status == RMS$_RNF)
         printf("RMSEXP - specified employee does not \
exist.\n");

      else
         {
❻            rms_status = sys$delete(&rab);
            if (rms_status != RMS$_NORMAL)
               error_exit("$DELETE");
         }
}
```

Key to *Example 2.6, "Utility Function: Deleting Records"*:

❶ A `do` loop prompts you to type a social security number at the terminal and places the response in the response buffer.

❷ The social security number is padded with blanks.

❸ Some members in the record access block must be initialized before the program can locate the record. Here, the key of reference (0 specifies the primary key), the location and size of the search string (this is the address of the response buffer and its size), and the type of record access (in this case, keyed access) are given.

❹ The RMS sys$find function locates the record specified by the social security number entered from the terminal.

❺ The program checks the `rms_status` variable for the values RMS$_NORMAL and RMS$_RNF (record not found). A message is displayed if the record cannot be found. Any other error is a fatal error.

❻ The RMS sys$delete function deletes the record. The return status is checked only for success.

Example 2.7, "Utility Function: Typing the File" shows the function that displays the employee file at the terminal. This function is called from the `main` function when '`t`' or '`T`' is entered in response to the menu.

Example 2.7. Utility Function: Typing the File

```
/* This segment of RMSEXP.C contains the function that    *
 * displays a single record at the terminal.              */

void type_employees(void)
```

```
{
❶   int  number_employees;

❷   rab.rab$b_krf =  1;

❸   rms_status =  sys$rewind(&rab);
   if (rms_status != RMS$_NORMAL)
      error_exit("$REWIND");

❹   printf("\n\nEmployees (Sorted by Last Name)\n\n");
   printf("Last Name      First Name          SSN      \
          Comments\n");

   printf("---------      ----------          ---------\
          --------\n\n");
❺  rab.rab$b_rac =  RAB$C_SEQ;
   rab.rab$l_ubf =  (char *) &record;
   rab.rab$w_usz =  RECORD_SIZE;

❻   for(number_employees = 0; ; number_employees++)
      {
         rms_status =  sys$get(&rab);
         if (rms_status != RMS$_NORMAL && rms_status !=
                                          RMS$_EOF)
            error_exit("$GET");
         else
            if (rms_status == RMS$_EOF)
               break;

         printf("%.*s%.*s%.*s%.*s\n",
                 SIZE_LNAME, record.last_name,
                 SIZE_FNAME, record.first_name,
                 SIZE_SSN, record.ssn,
                 SIZE_COMMENTS, record.comments);
      }

❼   if (number_employees)
      printf("\nTotal number of employees = %d.\n",
                number_employees);
   else
      printf("[Data file is empty.]\n");
}
```

Key to *Example 2.7, "Utility Function: Typing the File"*:

❶ A running total of the number of records in the file is kept in the `number_employees` variable.

❷ The key of reference is changed to the alternate key so that the employees are displayed in alphabetical order by last name.

❸ The file is positioned to the beginning of the first record according to the new key of reference, and the return status of the sys$rewind function is checked for success.

❹ A heading is displayed.

❺ Sequential record access is specified, and the location and size of the record is given.

❻ A `for` loop controls the following operations:

- Incrementing the `number_employees` counter
- Locating a record and placing it in the record structure, using the RMS sys$get function
- Checking the return status of the RMS sys$get function
- Displaying the record at the terminal

❼ This `if` statement checks for records in the file. The result is a display of the number of records or a message indicating that the file is empty.

Example 2.8, "Utility Function: Printing the File" shows the function that prints the file on the printer. This function is called by the `main` function when 'p' or 'P' is entered in response to the menu.

Example 2.8. Utility Function: Printing the File

```
/* This segment of RMSEXP.C contains the function that    *
 * prints the file.                                       */

void print_employees(void)
{
   int  number_employees;
   FILE *fp;

❶   fp = fopen("personnel.lis", "w", "rat=cr",
                      "rfm=var", "fop=spl");
   if (fp == NULL)
      {
         perror("RMSEXP - failed opening listing \
file");

         exit(SS$_NORMAL);
      }

❷   rab.rab$b_krf =  0;

❸   rms_status =  sys$rewind(&rab);
   if (rms_status != RMS$_NORMAL)
      error_exit("$REWIND");

❹   fprintf(fp,"\n\nEmployees (Sorted by SSN)\n\n");
   fprintf(fp,"Last Name      First Name       SSN      \
     Comments\n");

   fprintf(fp,"---------      ----------       ---------\
     --------\n\n");

❺   rab.rab$b_rac =  RAB$C_SEQ;
   rab.rab$l_ubf =  (char *) &record;
   rab.rab$w_usz =  RECORD_SIZE;

❻   for(number_employees = 0; ; number_employees++)
      {
         rms_status =  sys$get(&rab);
         if (rms_status != RMS$_NORMAL &&
             rms_status != RMS$_EOF)
            error_exit("$GET");
```

```
         else
            if (rms_status == RMS$_EOF)
               break;

         fprintf(fp, "%.*s%.*s%.*s%.*s",
                 SIZE_LNAME,record.last_name,
                 SIZE_FNAME,record.first_name,
                 SIZE_SSN,record.ssn,
                 SIZE_COMMENTS,record.comments);
      }
❼   if (number_employees)
      fprintf(fp, "Total number of employees = %d.\n",
              number_employees);
   else
      fprintf(fp,"[Data file is empty.]\n");

❽   fclose(fp);
      printf("[Listing file\"personnel.lis\"spooled to \
SYS$PRINT.]\n");
}
```

Key to *Example 2.8, "Utility Function: Printing the File"*:

❶ This function creates a sequential file with carriage-return carriage-control, variable-length records. It spools the file to the printer when the file is closed. The file is created using the standard I/O library function `fopen`, which associates the file with the file pointer, `fp`.

❷ The key of reference for the indexed file is the primary key.

❸ The RMS sys$rewind function positions the file at the first record. The return status is checked for success.

❹ A heading is written to the sequential file using the standard I/O library function `fprintf`.

❺ The record access, user buffer address, and user buffer size members of the record access block are initialized for keyed access to the record located in the record structure.

❻ A `for` loop controls the following operations:

- Initializing the running total and then incrementing the total at each iteration of the loop
- Locating the records and placing them in the record structure with the RMS sys$get function, one record at a time
- Checking the `rms_status` information for success and end-of-file
- Writing the record to the sequential file

❼ The `number_employees` counter is checked. If it is 0, a message is printed indicating that the file is empty. If it is not 0, the total is printed at the bottom of the listing.

❽ The sequential file is closed. Since it has the spl record attribute, the file is automatically spooled to the printer. The function displays a message at the terminal stating that the file was successfully spooled.

Example 2.9, "Utility Function: Updating the File" shows the function that updates the file. This function is called by the `main` function when '`u`' or '`U`' is entered in response to the menu.

Example 2.9. Utility Function: Updating the File

```
/* This segment of RMSEXP.C contains the function that    *
 * updates the file.                                      */

void update_employee(void)
{
   int i;
❶   do
      {
         printf("(UPDATE) Enter Social Security Number\
      ");

         gets(response);
         i =  strlen(response);
      }
   while(i == 0);

❷   while(i < SIZE_SSN)
      response[i++] = ' ';

❸  rab.rab$b_krf =  0;
   rab.rab$l_kbf =  response;
   rab.rab$b_ksz =  SIZE_SSN;
   rab.rab$b_rac =  RAB$C_KEY;
   rab.rab$l_ubf =  (char *) &record;
   rab.rab$w_usz =  RECORD_SIZE;

❹   rms_status =  sys$get(&rab);


   if (rms_status != RMS$_NORMAL && rms_status != RMS$_RNF)
      error_exit("$GET");
   else
      if (rms_status == RMS$_RNF)
         printf("RMSEXP - specified employee does not \
exist.\n");

❺      else
         {
            printf("Enter the new data or RETURN to leave \
data unmodified.\n\n");

            printf("Last Name:");
            gets(response);
            if (strlen(response))
               strncpy(record.last_name, response,
                       SIZE_LNAME);

            printf("First Name:");
            gets(response);
            if (strlen(response))
               strncpy(record.first_name, response,
                       SIZE_FNAME);

            printf("Comments:");
            gets(response);
            if (strlen(response))
```

```
                strncpy(record.comments, response,
                        SIZE_COMMENTS);

❻             pad_record();

❼             rms_status =  sys$update(&rab);
              if (rms_status != RMS$_NORMAL)
                 error_exit("$UPDATE");

              printf("[Record has been successfully \
updated.]\n");
           }
}
```

Key to *Example 2.9, "Utility Function: Updating the File"*:

❶ A `do` loop prompts for the social security number and places the response in the response buffer.

❷ The response is padded with blanks so that it will correspond to the field in the file.

❸ Some of the members in the record access block are initialized for the operation. The primary key is specified as the key of reference, the location and size of the key value are given, keyed access is specified, and the location and size of the record are given.

❹ The RMS sys$get function locates the record and places it in the record structure. The function checks the `rms_status` value for RMS$_NORMAL and RMS$_RNF (record not found). If the record is not found, a message is displayed. If the record is found, the program prints instructions for updating the record.

❺ If you press the Return key, the record is placed in the record structure unchanged. If you make a change to the record, the new information is placed in the record structure.

❻ The fields in the record are padded with blanks.

❼ The RMS sys$update function rewrites the record. The program then checks that the update operation was successful. Any error causes the program to call the fatal error-handling routine.

Chapter 3. Using VSI C in the Common Language Environment

This chapter discusses the following topics:

- OpenVMS calling standard conventions (*Section 3.1, "Basic Calling Standard Conventions"*)
- Parameter-passing mechanisms (*Section 3.2, "Specifying Parameter-Passing Mechanisms"*)
- Interlanguage calling (*Section 3.3, "Interlanguage Calling"*)
- Sharing global data (*Section 3.4, "Sharing Global Data"*)
- OpenVMS Run-Time Library (RTL) routines (*Section 3.5, "OpenVMS Run-Time Library Routines"*)
- OpenVMS system services routines (*Section 3.6, "OpenVMS System Services Routines"*)
- Calling routines (*Section 3.7, "Calling Routines"*)
- Variable-length argument lists in system services (*Section 3.8, "Variable-Length Argument Lists in System Services"*)
- Return status values (*Section 3.9, "Return Status Values"*)
- Examples of calling system routines (*Section 3.10, "Examples of Calling System Routines"*)

The VSI C compiler is part of the OpenVMS common language environment. This environment defines certain calling procedures and guidelines that allow you to call routines written in different languages from VSI C programs, to call VSI C functions from programs written in other languages, or to call prewritten system routines from VSI C programs. You can call any one of the following routine types from VSI C:

- Routines written in other OpenVMS languages
- OpenVMS RTL routines
- OpenVMS system services
- OpenVMS utility routines

The terms routine, procedure, and function are used throughout this chapter. A *routine* is a closed, ordered set of instructions that performs one or more specific tasks. Every routine has an entry point (the routine name), and optionally an argument list. Procedures and functions are specific types of routines: a *procedure* is a routine that does not return a value; a *function* is a routine that returns a value by assigning that value to the function's identifier.

System routines are prewritten OpenVMS routines that perform common tasks, such as finding the square root of a number or allocating virtual memory. You can call any system routine from your program, provided that VSI C supports the data structures required to call the routine. The system routines used most often are OpenVMS RTL routines and system services. System routines, which are discussed later in this chapter, are documented in detail in the *VMS Run-Time Library Routines Volume* and the *VSI OpenVMS System Services Reference Manual*.

3.1. Basic Calling Standard Conventions

The *VSI OpenVMS Calling Standard* describes the concepts used by all OpenVMS languages to invoke routines and pass data between them. It also describes the differences between the VAX and Alpha parameter-passing mechanisms. The OpenVMS calling standard specifies the following attributes:

- Register usage
- Stack usage
- Function return value
- Argument list

The following sections discuss these attributes in more detail for OpenVMS VAX systems. For more detail on OpenVMS Alpha systems, see the *VSI OpenVMS Calling Standard*.

The calling standard also defines such attributes as the calling sequence, the argument data types and descriptor formats, condition handling, and stack unwinding. These attributes are discussed in detail in the *OpenVMS Programming Interfaces: Calling a System Routine*.

3.1.1. Register and Stack Usage

The calling standard defines several registers and their uses, as listed in *Table 3.1, "VAX Register Usage"* for VAX systems and *Table 3.2, "Alpha Register Usage"* for Alpha systems.

Table 3.1. VAX Register Usage

Register	Use
PC	Program counter
SP	Stack pointer
FP	Current stack frame pointer
AP	Argument pointer
R1	Environment value (when necessary)
R0, R1	Function return value registers

Table 3.2. Alpha Register Usage

Register	Use
PC	Program counter
SP	Stack pointer
FP	Frame pointer for current procedure
R25	Argument information register
R16 to R21, F16 to F21	Argument list registers
R0	Function return value register

By definition, any called routine can use registers R2 through R11 for computation, and the AP register as a temporary register.

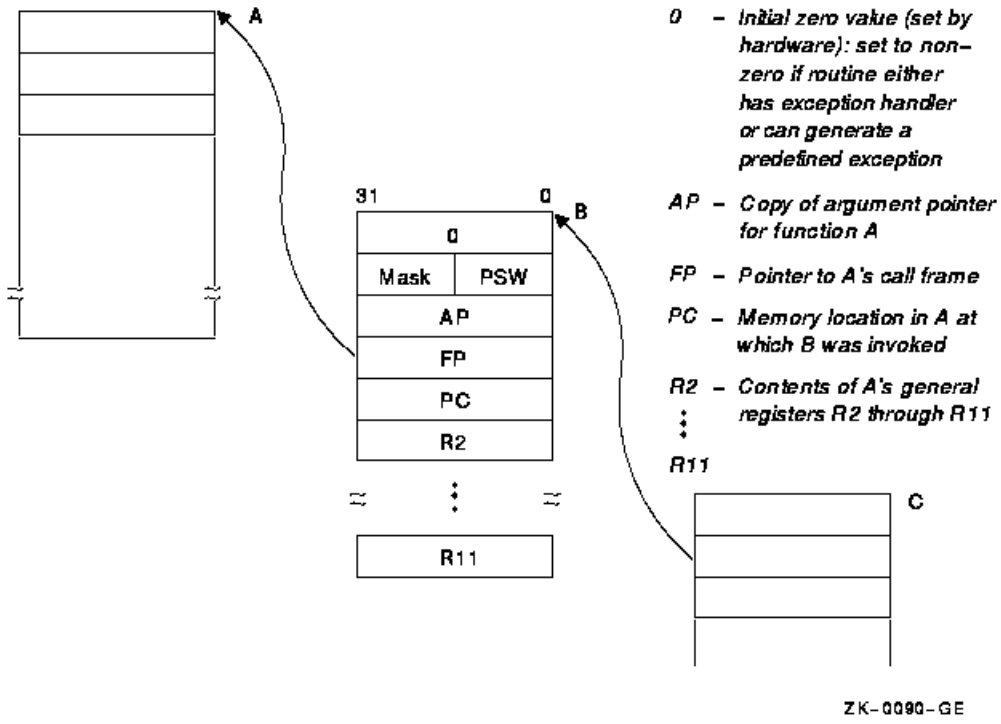
In the calling standard, a *stack* is defined as a last-in/first-out (LIFO) temporary storage area that the system allocates for every user process. The system keeps information about each routine call in the current image on the call stack. Then, each time you call a routine, the system creates a structure on this call stack, known as the *call frame*. The call frame for each active process contains the following data:

- A pointer to the call frame of the previous routine call. This pointer corresponds to the frame pointer (FP).
- The argument pointer (AP) of the previous routine call.
- The storage address of the point at which the routine was called; that is, the address of the instruction following the call to the current routine. This address is called the program counter (PC).
- The contents of other general registers. Based on a mask specified in the control information, the system restores the saved contents of these registers to the calling routine when control returns to it.

When a routine completes execution, the system uses the frame pointer in the call frame of the current routine to locate the frame of the previous routine. The system then removes the call frame of the current routine from the stack.

Figure 3.1, "The Call Stack" shows the call stack and several call frames for VAX processors. Function A calls function B, which calls function C. When a function reaches a `return` statement or when control reaches the end of the function, the system uses the frame pointer in the call frame of the current function to locate the frame of the previous function. It then removes the call frame of the current function from the stack.

Figure 3.1. The Call Stack



3.1.2. Return of the Function Value

A function is a routine that returns a single value to the calling routine. The *function value* represents the value of the expression in the `return` statement. According to the calling standard, a function value may be returned as either an actual value or a condition value that indicates success or failure.

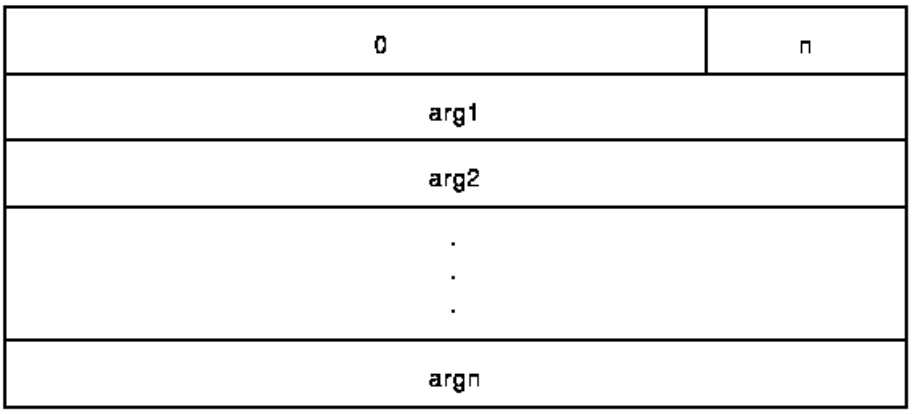
3.1.3. The Argument List

The *VSI OpenVMS Calling Standard* also defines a data structure called the *argument list*. You use an argument list to pass information to a routine and receive results.

On OpenVMS Alpha systems, an argument list is formed using registers R16 to R21 or F16 to F21, and a collection of quadwords in memory (depending on the number and type of the arguments).

On OpenVMS VAX systems, an argument list is a collection of longwords in memory that represents a routine parameter list and possibly includes a function value. *Figure 3.2, "Structure of an OpenVMS VAX Argument List"* shows the structure of a typical OpenVMS VAX argument list.

Figure 3.2. Structure of an OpenVMS VAX Argument List

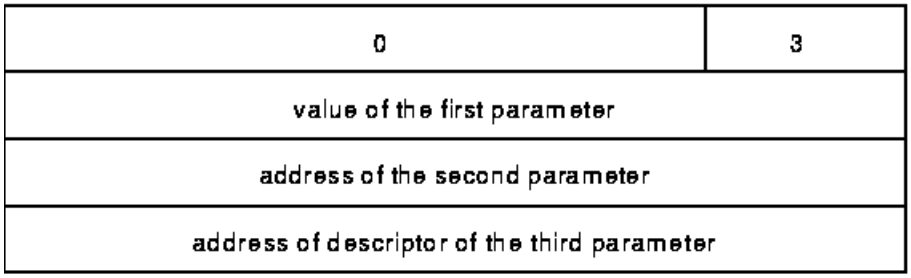


The first longword must be present; this longword stores the number of arguments (the argument count: *n*) as an unsigned integer value in the low byte of the longword with a maximum of 255 arguments. The remaining 24 bits of the first longword are reserved for use by VSI and should be 0. The longwords labeled *arg1* through *argn* are the actual parameters, which can be any of the following addresses or value:

- An uninterpreted 32-bit value that is passed by value
- An address that is passed by reference
- An address of a descriptor that is passed by descriptor

The argument list contains the parameters that are passed to the routine. Depending on the passing mechanisms for these parameters, the forms of the arguments contained in the argument list vary. For example, if you pass three arguments, the first by value, the second by reference, and the third by descriptor, the argument list would contain the value of the first argument, the address of the second, and the address of the descriptor of the third. *Figure 3.3, "Example of an OpenVMS VAX Argument List"* shows this argument list.

Figure 3.3. Example of an OpenVMS VAX Argument List



For additional information on the OpenVMS calling standard, see the *VSI OpenVMS Calling Standard*.

3.2. Specifying Parameter-Passing Mechanisms

When you pass data between routines that are not written in the same OpenVMS language, you have to specify how you want that data to be represented and interpreted. You do this by specifying a *parameter-passing mechanism*.

The calling standard defines three ways to pass data in an argument list. When you code a reference to a non-VSI C procedure, you must know how to pass each argument and write the function reference accordingly.

The following list describes the three argument-passing mechanisms:

- By immediate value

  When an argument is passed by immediate value, the actual value of the argument is present in the argument list. This is the default argument-passing mechanism for all function references written in VSI C.

- By reference

  When an argument is passed by reference, the address of the argument is present in the argument list. Use the C ampersand operator (&) to pass the address of an argument, or pass a pointer to the argument by value.

- By descriptor

  When an argument is passed by descriptor, the address of a data structure describing the argument is present in the argument list. From a VSI C program, you pass a descriptor first by creating a structure (`struct`) that meets the descriptor requirements of the called procedure and then by passing the structure's address with the ampersand operator or by passing a pointer to that structure by value.

The following sections outline each of these parameter-passing mechanisms in more detail.

3.2.1. Passing Arguments by Immediate Value

By default, all values or expressions in a VSI C function's argument list are passed by immediate value (except for X_FLOATING on OpenVMS Alpha systems, which is passed by reference). The expressions are evaluated and the results placed directly in the argument list of the CALL machine instruction.

The following statement declares the entry point of the Set Event Flag SYS$SETEF system service, which is used to set a specific event flag to 1:

```
/*  Declare the function as a function returning type int.      */

int  SYS$SETEF();
```

The SYS$SETEF system service call requires one argument—the number of the event flag to be set—to be passed by immediate value. VSI C for OpenVMS systems converts linker-resolved variable names (such as the entry-point names of system service calls) to uppercase. You do not have to declare them in uppercase in your program (unless you compile your module with /NAMES=AS_IS). However, linker-resolved variable names must be declared and used with identical cases in each module. The documentation uses uppercase as a convention for referring to system service calls to highlight them in the text and examples.

VSI C does not require you to declare a function or to specify the number or types of the function's arguments. However, if you call a function without declaring it or without providing argument information in the declaration, VSI C does not check the types of the arguments in a call to that function. If you declare a function prototype, the compiler does check the arguments in a call to make sure that they have the same type. (See the *VSI C Reference Manual* [https://docs.vmssoftware.com/vsi-c-language-reference-manual/] for more information on function prototypes.)

Like all system services, SYS$SETEF returns an integer value (the return status of the service) in register 0. Most system services return an integer completion status; therefore, the system service does not always have to be declared before it is used. The examples in this chapter declare system services for completeness.

In the *VSI OpenVMS System Services Reference Manual*, you can find the specification of each service's arguments. SYS$SETEF, for example, takes one argument, an event flag number. It returns one of four status values, which are represented by the symbolic constants shown in *Table 3.3, "Status Values of* SYS$SETEF*"*.

Table 3.3. Status Values of SYS$SETEF

Returned	Status	Description
SS$_WASCLR	Success	Flag was previously clear
SS$_WASSET	Success	Flag was previously set
SS$_ILLEFC	Failure	Illegal event flag number
SS$_UNASEFC	Failure	Event flag not in associated cluster

The system services manual also defines event flags as integers in the range 0 to 127, grouped in clusters of 32. Clusters 0 and 1, comprising flags 0 to 31 and 32 to 63, respectively, are local clusters available to any process, with the restriction that flags 24 to 31 are reserved for use by the OpenVMS system. There are many ways of passing valid event flag numbers from your VSI C program to SYS$SETEF. One way is to use `enum` to define a subset of integers, as follows:

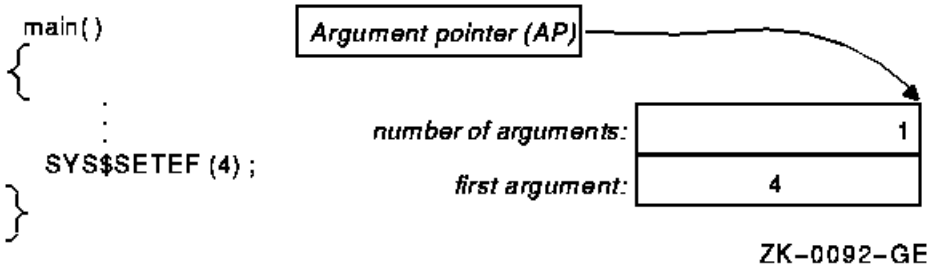
```
enum cluster0 {completion, breakdown, beginning} event;
```

After the flag numbers are defined, call the SYS$SETEF service with the following code:

```
.
.
.
int  status;
event = completion;
.
.
.
status = SYS$SETEF(event);          /* Set event flag.          */
.
.
.
```

Figure 3.4, "Passing Arguments by Immediate Value" shows an argument being passed by immediate value; in this case, the event flag number passed to SYS$SETEF.

Figure 3.4. Passing Arguments by Immediate Value



Since argument lists consist of longwords, the calling standard dictates that immediate-value arguments be expressed in 32 bits. A single-precision, floating-point (F_floating) value is only 32 bits long, but the compiler promotes all arguments of type `float` to `double` (64 bits on a VAX processor) unless a function prototype declaration is used for the called function. This double-precision value is passed as two immediate values (two longwords).

Note

The passing of double-precision immediate values is a violation of the calling standard for OpenVMS VAX systems, but is an allowed exception for VSI C.

On rare occasions, the `float`-to-`double` promotion requires some additional programming. For instance, the function OTS$POWRJ, in the VAX Common Run-Time Procedure Library, computes the value of a floating-point number raised to the power of a signed longword (in C terms, a `float` to the power of an `int`). This function (and others like it) is called implicitly by high-level OpenVMS languages that have an exponentiation operator as part of the language. It requires that both its arguments be passed as immediate values, and it returns a single-precision (`float`) result. To pass a floating-point base to the procedure, you must use some method to avoid promoting `float` arguments. The recommended method is to declare the procedure using a function prototype declaration, as shown in *Example 3.1, "Passing Floating-Point Arguments by Immediate Value"*.

Example 3.1. Passing Floating-Point Arguments by Immediate Value

```
/*  This program shows how to pass a floating-point value,     *
 *  using prototypes to avoid promoting floating               *
 *  arguments to arguments of type double.                     */
```

```
#include <stdio.h>

/*  This declared function returns a value of type float.  It  *
 *  should be called as follows:  OTS$POWRJ(base, power),      *
 *  where base is of type float and power is of type int.      */

float OTS$POWRJ(float, int);

main(void)
{
                                    /*  To hold result of       *
                                     *   OTS$POWRJ              */
   float  result;
   int    power;                    /*  Power argument          */

   float base;

   base =  3.145;                   /*  Assign constant to base */
   power  =  2;
   result =  OTS$POWRJ(base, power);

   printf("Result= %f\n", result);
}
```

Note

To get the correct results on IA-64 systems, compile the preceding example with /FLOAT=G_FLOAT.

The example does not show the methods for handling arithmetic errors that result from the operation performed. For more information on error handling in this context, and on the run-time library in general, see the *VMS Run-Time Library Routines Volume*.

When you pass a parameter by value, you pass a copy of the parameter value to the routine instead of passing its address. Because the actual value of the parameter is passed, the routine does not have access to the storage location of the parameter; therefore, any changes that you make to the parameter value in the routine do not affect the value of that parameter in the calling routine.

3.2.2. Passing Arguments by Reference

Some system services and run-time library procedures expect arguments passed by reference. This means that the argument list contains the address of the argument rather than its value. This mechanism is also used by default by some programming languages, such as PL/I, and is available as an option in others, such as Pascal.

In C, you can use the ampersand operator (&) to pass an argument by reference; that is, the ampersand operator causes the argument's address to be passed. Note that an array name without brackets or a function name without parentheses in an argument list always results in passing the address of the array or function; the ampersand is unnecessary. You can also pass a pointer by value, which is the same as passing the item it points to by reference.

In the special case of argument lists, VSI C in VAX C mode allows the ampersand operator to be used on constants as well. You should limit this use of the ampersand solely to calls to OpenVMS system functions to ensure portability of your VSI C programs to other C compilers.

For example, the Read Event Flags (SYS$READEF) system service requires that its first argument be passed by immediate value and its second argument be passed by reference. SYS$READEF returns the

status of all the event flags in a particular cluster. (Event flags are numbered from 0 to 127 and arranged in clusters of 32, such that flags 0 to 31 comprise cluster 0, flags 32 to 63, cluster 1, and so forth.)

The first SYS$READEF argument is any event flag number in the cluster of interest. The second argument is the address of a longword that receives the status of all 32 event flags in that cluster. In addition to the event-flag status value, the system service returns one of the status values shown in *Table 3.4, "Status Values of* SYS$READEF*"* expressed as a global symbol.

Table 3.4. Status Values of SYS$READEF

Returned	Status	Description
SS$_WASCLR	Success	Specified event flag was clear
SS$_WASSET	Success	Specified event flag was set
SS$_ACCVIO	Failure	Could not write to status longword
SS$_ILLEFC	Failure	Event flag number was illegal
SS$_UNASEFC	Failure	Cluster of interest not accessible

Example 3.2, "Passing Arguments by Reference" shows a call to the SYS$READEF system service from a VSI C program.

Example 3.2. Passing Arguments by Reference

```
/*  This program shows how to call system service SYS$READEF.  */

#include <ssdef.h>
#include <stdio.h>

int  SYS$READEF();

main(void)
{
                                    /*  Longword that receives   *
                                     *   the status of the       *
                                     *   event flag cluster.     */
   unsigned cluster_status;

   int return_status;               /* Status: SYS$READEF.       */

                                    /*  Argument values for      *
                                     *   SYS$READEF.             */
   enum  cluster0
      {
         completion, breakdown, beginning
      }  event;
      .
      .
      .
   event = completion;              /*  Event flag in cluster 0. */

                                    /*  Obtain status of         *
                                     *   cluster 0.  Pass value  *
                                     *   of event and address    *
                                     *   of cluster_status.      */
```

```
   return_status =  SYS$READEF(event, &cluster_status);

                                  /*  Check for successful    *
                                   *   call                   */
   if (return_status != SS$WASCLR && return_status != SS$WASSSET)
      {
         /* Handle the error here.                           */
            .
            .
            .
      }
   else
      {
         /*  Check bits of interest in cluster_status here.  */
            .
            .
            .
      }
}
```

3.2.3. Passing Arguments by Descriptor

A *descriptor* is a structure that describes the data type, size, and address of a data structure. According to the *VSI OpenVMS Calling Standard*, you must pass a descriptor by placing its address in the argument list. To pass an argument by descriptor from a VSI C program, perform the following steps:

1. Write a structure declaration that models the required descriptor. This involves including the `<descrip.h>` header file to define `struct` tags for all the forms of descriptors.
2. Assign appropriate values to the structure members.
3. Use the structure name, with an ampersand operator (&) in the function reference, to put the structure's address in the argument list.

VSI C never passes arguments by descriptor by default; you must take explicit action to pass an argument by descriptor. Also, if you write structure or union names in a function's argument list without the ampersand operator, the structure or union is passed by immediate value to the called function. You pass arguments by descriptor only when the called function is written in another language and explicitly requires this mechanism.

Note

The passing of structures as immediate values can be a violation of the OpenVMS calling standard if the entire structure is larger than one longword of memory. This type of argument passing is an allowed exception for VSI C.

There are several classes of descriptor. Each class requires that certain bits be set in the first longword of the descriptor. For more information about the descriptors and their formats, see the *OpenVMS Programming Interfaces: Calling a System Routine*. You can model descriptors in VSI C as follows:

```
struct  dsc$descriptor
   {
      unsigned  short  dsc$w_length; /*  Length of data         */
      char  dsc$b_dtype              /*  Data type code         */
      char  dsc$b_class              /*  Descriptor class code  */
      char  *dsc$a_pointer           /*  Address of first byte  */
```

```
};
```

In this model, the variable `dsc$w_length` is a 16-bit word containing the length of the entire data; the unit (for example, bit or byte) in which the length is measured depends on the descriptor class. The member `dsc$b_dtype` is a byte containing a numeric code; the code denotes the data type of the data. The class member `dsc$b_class` is another byte code giving the descriptor class. *Table 3.5, "Valid Class Codes"* shows the valid class codes.

Table 3.5. Valid Class Codes

Class Code	Symbolic Name	Descriptor Class
1	DSC$K_CLASS_S	Scalar, string
2	DSC$K_CLASS_D	Dynamic string descriptor
3	—	Reserved by VSI
4	DSC$K_CLASS_A	Array
5	DSC$K_CLASS_P	Procedure
6	DSC$K_CLASS_PI	Procedure incarnation
7	DSC$K_CLASS_J	Reserved by VSI
8	DSK$K_CLASS_JI	This is obsolete
9	DSC$K_CLASS_SD	Decimal scalar string
10	DSC$K_CLASS_NCA	Noncontiguous array
11	DSC$K_CLASS_VS	Varying string
12	DSC$K_CLASS_VSA	Varying string array
13	DSC$K_CLASS_UBS	Unaligned bit string
14	DSC$K_CLASS_UBA	Unaligned bit array
15	DSC$K_CLASS_SB	String with bounds descriptor
16	DSC$K_CLASS_UBSB	Unaligned bit string with bounds descriptor
17-190	—	Reserved by VSI
191	DSC$K_CLASS_BFA	Basic file array
192-255	—	Reserved for customer applications

The atomic data types shown in *Table 3.6, "Atomic Data Types"* are supported by VSI C; all others are not directly supported by the language. See the *OpenVMS Programming Interfaces: Calling a System Routine* manual for a complete list of atomic class codes.

Table 3.6. Atomic Data Types

Class Code	Symbolic Name	Descriptor Class
2	DSC$K_DTYPE_BU	Byte (unsigned)
3	DSC$K_DTYPE_WU	Word (unsigned)
4	DSC$K_DTYPE_LU	Longword (unsigned)

Class Code	Symbolic Name	Descriptor Class
6	DSC$K_DTYPE_B	Byte integer (signed)
7	DSC$K_DTYPE_W	Word integer (signed)
8	DSC$K_DTYPE_L	Longword integer (signed)
10	DSC$K_DTYPE_F	F_floating
11	DSC$K_DTYPE_D	D_floating
14	DSC$K_DTYPE_T	Character string
27	DSC$K_DTYPE_G	G_floating
52	DSC$K_DTYPE_FS	IEEE S_floating
53	DSC$K_DTYPE_FT	IEEE T_floating

The last member of the structure model, `dsc$a_pointer`, points to the first byte of the data.

To pass an argument by descriptor, you define and assign values to the data following normal C programming practices. You must define a `dsc$descriptor` structure and assign the data's address to the `dsc$a_pointer` member. You must also assign appropriate values to the members `dsc$w_length`, `dsc$b_dtype`, and `dsc$b_class`. For the specific requirements of each descriptor class, see the *OpenVMS Programming Interfaces: Calling a System Routine* manual.

For example, the Set Process Name (SYS$SETPRN) system service, which enables a process to establish or change its process name, accepts a process name as a fixed-length character string passed by descriptor. The character string can have from 1 to 15 characters. The system service returns status values that are represented by the symbolic constants shown in *Table 3.7, "Status Values of* SYS$SETPRN*"*.

Table 3.7. Status Values of SYS$SETPRN

Returned	Status	Description
SS$_NORMAL	Success	Normal completion
SS$_ACCVIO	Failure	Inaccessible descriptor
SS$_DUPLNAM	Failure	Duplicate process name
SS$_IVLOGNAM	Failure	Invalid length

Example 3.3, "Passing Arguments by Descriptor" shows a call to this system service from a VSI C program.

Example 3.3. Passing Arguments by Descriptor

```
/*  This program shows a call to system service SYS$SETPRN.    */

#include <ssdef.h>
#include <stdio.h>
                                    /*  Define structures for   *
                                     *   descriptors            */
#include <descrip.h>

int SYS$SETPRN();
```

```
int main(void)
{
   int  ret;                          /*  Define return status of *
                                       *   SYS$SETPRN             */

                                      /*  Name the descriptor     */
   struct  dsc$descriptor_s  name_desc;

   char  *name =  "NEWPROC";          /*  Define new process name */
      .
      .
      .
                                      /*  Length of name WITHOUT  *
                                       *   null terminator        */
   name_desc.dsc$w_length = strlen(name);

                                      /*  Put address of          *
                                       *   shortened string in    *
                                       *   descriptor             */
   name_desc.dsc$a_pointer =  name;

                                      /*  String descriptor class */
   name_desc.dsc$b_class =  DSC$K_CLASS_S;

                                      /*  Data type: ASCII string */
   name_desc.dsc$b_dtype =  DSC$K_DTYPE_T;
      .
      .
      .
   ret =  SYS$SETPRN(&name_desc);

   if (ret != SS$_NORMAL)             /*  Test return status      */
      fprintf(stderr, "Failed to set process name\n"),
      exit(ret);
      .
      .
      .
}
```

In *Example 3.3, "Passing Arguments by Descriptor"*, the call to SYS$SETPRN must use the ampersand operator; otherwise, `name_desc`, rather than its address, is passed.

Although this example explicitly sets individual fields in its `name_desc` string descriptor, in practice, the run-time initialization of compile-time constant string descriptors is not performed in this manner. Instead, the fields of compile-time constant descriptors are usually initialized with initialized structures of storage class `static`.

For the purpose of string descriptor initialization, VSI C provides a simple preprocessor macro in the `<descrip.h>` header file. This macro is named $DESCRIPTOR. It takes two arguments, which it uses in a standard VSI C structure declaration. The first argument is an identifier specifying the name of the descriptor to be declared and initialized. The second argument is a pointer to the data byte to be used as the value of the descriptor. Since a character-string constant is interpreted as an initialized pointer to `char`, you may specify the second argument as a simple string constant. You may use the $DESCRIPTOR macro in any context where a declaration may be used. The scope of the declared string descriptor identifier name is identical to the scope of a simple `struct` definition as expanded by the macro.

Example 3.4, "Passing Compile-Time String Descriptors" shows a variant of the program in *Example 3.3, "Passing Arguments by Descriptor"*. Here, the $DESCRIPTOR macro is used to create a compile-time string descriptor and to pass it to the SYS$SETPRN system service routine. In Example *Example 3.4, "Passing Compile-Time String Descriptors"*, the program returns the status value returned by SYS$SETPRN to DCL for interpretation.

Example 3.4. Passing Compile-Time String Descriptors

```
/*  This program returns the status value returned by          *
 *  SYS$SETPRN.                                                */

#include  <descrip.h>            /*  Define $DESCRIPTOR macro. */

int  SYS$SETPRN();

int main(void)
{
                              /*  Initialize structure name_desc *
                               *  as string descriptor.          */
   static  $DESCRIPTOR(name_desc,"NEWPROC");

   return  SYS$SETPRN(&name_desc);
}
```

To test the results of the preceding example, do the following:

```
$ SHOW PROCESS       ! Note the process name.

$ RUN example        ! Run the example.

$ SHOW PROCESS       ! Note that the process name has changed.
```

The $DESCRIPTOR macro is used in further examples in this chapter.

3.2.4. VSI C Default Parameter-Passing Mechanisms

There are default parameter-passing mechanisms established for every data type you can use with VSI C. *Table 3.8, "Valid Parameter-Passing Mechanisms in VSI C"* lists the VSI C data types you can use with each parameter-passing mechanism. Asterisks appear next to the default parameter-passing mechanism for that particular data type.

Table 3.8. Valid Parameter-Passing Mechanisms in VSI C

Data Type	By Reference	By Descriptor	By Value
Variables	Yes	Yes	Yes*
Constants	Yes (VAX C mode only)	Yes	Yes*
Expressions	No	No	Yes*
Array elements	Yes	Yes	Yes*
Entire array	Yes*	Yes	No
String constants	Yes*	Yes	No
Structures and unions	Yes	Yes	Yes*

Data Type	By Reference	By Descriptor	By Value
Functions	Yes*	Yes	No

You must use the appropriate parameter-passing mechanisms whenever you call a routine written in some other OpenVMS language or some prewritten system routine.

3.3. Interlanguage Calling

In VSI C, you can call external routines written in other languages or VSI C routines from routines written in other languages as either functions or subroutines. When you call an external routine as a function, a single value is returned. When you call an external routine as a subroutine (a `void` function), any values are returned in the argument list.

By default, VSI C passes all arguments by immediate value with the exception of arrays and functions; these are passed by reference. *Table 3.9, "Default Passing Mechanisms"* lists the default passing mechanisms for other OpenVMS languages.

Table 3.9. Default Passing Mechanisms

Language	Arrays	Numeric Data	Character Data
MACRO	No default	No default	No default
Pascal	Reference	Reference	Descriptor
BASIC	Descriptor	Reference	Descriptor
COBOL	N/A	Reference	Reference
FORTRAN	Reference	Reference	Descriptor

The following sections describe the methods involved in using VSI C with routines written in other OpenVMS languages.

3.3.1. Calling FORTRAN

When calling VSI Fortran from VSI C or vice versa, note these considerations. VSI Fortran argument lists and argument descriptors are usually allocated statically. When it is possible, and to optimize space and time, the VSI Fortran compiler pools the argument lists and initializes them at compile time. Sometimes several calls may use the same argument list.

In VSI C, you often use arguments as local variables, and modify them at will. If a VSI C routine that modifies an argument is called from a VSI Fortran routine, unintended and incorrect side effects may occur.

The following example shows a VSI C routine that is invalid when called from VSI Fortran:

```
void f(int *x)      /* x is a FORTRAN INTEGER passed by reference */

{
    /* The next assignment is OK.  It is permitted to modify what a
     * FORTRAN argument list entry points to. */
    *x = 0;                /* ok */

    /* The next assignment is invalid.  It is not permitted to modify
     * a FORTRAN argument list entry itself. */
    x = x + 1;               /* Invalid */
```

```
}
```

Another problem is the semantic mismatch between strings in C and strings in VSI Fortran. Strings in C vary in length and end in a null character. Strings in VSI Fortran do not end in a null character and are padded with spaces to some fixed length. In general, this mismatch means that strings may not be passed between VSI C and VSI Fortran unless you do additional work. You may make a VSI Fortran routine add a null character to a CHARACTER string before calling a VSI C function. You may also write code that explicitly gets the length of a VSI Fortran string from its descriptor and carefully pads the string with spaces after modifying it. An example later in this section shows a C function that carefully produces a proper string for VSI Fortran.

Example 3.5, "VSI C Function Calling a VSI Fortran Subprogram" shows a VSI C function calling a VSI Fortran subprogram with a variety of data types. For most scalar types, VSI Fortran expects arguments to be passed by reference but character data is passed by descriptor.

Example 3.5. VSI C Function Calling a VSI Fortran Subprogram

```
/*
 *  Beginning of VSI C function:
 */

#include <stdio.h>
#include <descrip.h>                /* Get layout of descriptors */

extern int fort();                  /* Declare FORTRAN function */

main(void)
{
    int i = 508;
    float f = 649.0;
    double d = 91.50;
    struct {
        short s;
        float f;
    } s = {-2, -3.14};
    auto $DESCRIPTOR(string1, "Hello, FORTRAN");
    struct dsc$descriptor_s string2;


    /* "string1" is a FORTRAN-style string declared and initialized using
 the
     * $DESCRIPTOR macro.  "string2" is also a FORTRAN-style string, but we
 are
     * declaring and initializing it by hand. */
    string2.dsc$b_dtype = DSC$K_DTYPE_T;  /* Type is CHARACTER */
    string2.dsc$b_class = DSC$K_CLASS_S;  /* String descriptor */
    string2.dsc$w_length = 3;             /* Three characters in string */
    string2.dsc$a_pointer = "bye";        /* Pointer to string value */

    printf("FORTRAN result is %d\n", fort(&i, &f, &d, &s, &string1,
 &string2));
}   /* End of VSI C function */


C
C     Beginning of FORTRAN subprogram:
C
```

```
      INTEGER FUNCTION FORT(I, F, D, S, STRING1, STRING2)
      INTEGER I
      REAL F
      DOUBLE PRECISION D
      STRUCTURE /STRUCT/
      INTEGER*2 SHORT
      REAL FLOAT
      END STRUCTURE
      RECORD /STRUCT/ S

C     You can tell FORTRAN to use the length in the descriptor
C     as done here for STRING1, or you can tell FORTRAN to ignore the
C     descriptor and assume the string has a particular length as done
C     for STRING2.  This choice is up to you.
      CHARACTER*(*) STRING1
      CHARACTER*3 STRING2


      WRITE(5, 10) I, F, D, S.SHORT, S.FLOAT, STRING1, STRING2
10    FORMAT(1X, I3, F8.1, D10.2, I7, F10.3, 1X, A, 2X, A)
      FORT = -15
      RETURN
      END
C     End of FORTRAN subprogram
```

Example 3.5, "VSI C Function Calling a VSI Fortran Subprogram" produces the following output:

```
508   649.0  0.92D+02     -2    -3.140 Hello, FORTRAN  bye
FORTRAN result is -15
```

Example 3.6, "VSI Fortran Subprogram Calling a VSI C Function" shows a VSI Fortran subprogram calling a VSI C function. Since the VSI C function is called from VSI Fortran as a subroutine and not as a function, the VSI C function is declared to have a return value of `void`.

Example 3.6. VSI Fortran Subprogram Calling a VSI C Function

```
C
C     Beginning of FORTRAN subprogram:
C
      INTEGER I
      REAL F(3)
      CHARACTER*10 STRING

C     Since this program does not have a C main program and you want
C     to use VSI C RTL functions from the C subroutine, you must call
C     DECC$CRTL_INIT to initialize the run-time library.
      CALL DECC$CRTL_INIT


      I = -617
      F(1) = 3.1
      F(2) = 0.04
      F(3) = 0.0016
      STRING = 'HELLO'

      CALL CSUBR(I, F, STRING)
      END
C     End of FORTRAN subprogram
```

```
/*
 *  Beginning of VSI C function:
 */
#include <stdio.h>
#include <descrip.h>                        /* Get layout of descriptors */

void csubr(int *i,                          /* FORTRAN integer, by reference
 */
    float f[3],                             /* FORTRAN array, by reference
 */
    struct dsc$descriptor_s *string)        /* FORTRAN character, by
 descriptor */
{
    int j;

    printf("i = %d\n", *i);

    for (j = 0; j < 3; ++j)
        printf("f[%d] = %f\n", j, f[j]);


    /* Since FORTRAN character data is not null-terminated, you must use
     * a counted loop to print the string.
     */
    printf("string = \"");
    for (j = 0; j < string->dsc$w_length; ++j)
        putchar(string->dsc$a_pointer[j]);
    printf("\"\n");

}   /*  End of VSI C function  */
```

Example 3.6, "VSI Fortran Subprogram Calling a VSI C Function" produces the following output:

```
i = -617
f[0] = 3.100000
f[1] = 0.040000
f[2] = 0.001600
string = "HELLO     "
```

Example 3.7, "VSI C Function Emulating a VSI Fortran CHARACTER(*) Function"* shows a C function that acts like a CHARACTER*(*) function in VSI Fortran.

Example 3.7. VSI C Function Emulating a VSI Fortran CHARACTER*(*) Function

```
C
C       Beginning of FORTRAN program:
C
        CHARACTER*9 STARS, C

C       Call a C function to produce a string of three "*" left-justified
C       in a nine-character field.
        C = STARS(3)

        WRITE(5, 10) C
10      FORMAT(1X, '"', A, '"')
        END
C       End of FORTRAN program
```

```
/*
 *  Beginning of VSI C function:
 */

#include <descrip.h>                          /* Get layout of descriptors */

/* Routine "stars" is equivalent to a FORTRAN function declared as
 * follows:
 *
 *        CHARACTER*(*) FUNCTION STARS(NUM)
 *        INTEGER NUM
 *

 * Note that a FORTRAN CHARACTER function has an extra entry added to
 * the argument list to represent the return value of the CHARACTER
 * function.  This entry, which appears first in the argument list,
 * is the address of a completely filled-in character descriptor.  Since
 * the C version of a FORTRAN character function explicitly uses this
 * extra argument list entry, the C version of the function is void!
 *

 * This example function returns a string that contains the specified
 * number of asterisks (or "stars").
 *
 */


void stars(struct dsc$descriptor_s *return_value, /* FORTRAN return value
 */
           int *num_stars)                 /* Number of "stars" to create
 */
{
    int i, limit;


    /* A FORTRAN string is truncated if it is too large for the memory area
     * allocated, and it is padded with spaces if it is too short. Set
     * limit to the number of stars to put in the string given the size
     * of the area used to store it. */
    if (*num_stars < return_value->dsc$w_length)
        limit = *num_stars;
    else
        limit = return_value->dsc$w_length;

    /* Create a string of stars of the specified length up to the limit of
       the string size. */
    for (i = 0; i < limit; ++i)
        return_value->dsc$a_pointer[i] = '*';

    /* Pad rest of string with spaces, if necessary. */
    for (; i < return_value->dsc$w_length; ++i)
        return_value->dsc$a_pointer[i] = ' ';
}  /*  End of VSI C Function   */
```

Example 3.7, "VSI C Function Emulating a VSI Fortran CHARACTER(*) Function"* produces the following output:

```
"***      "
```

3.3.2. Calling VAX MACRO

You can call a VAX MACRO routine from VSI C or vice versa. However, like all interlanguage calls, it is necessary to make sure that the actual arguments correspond to the expected formal parameter types. Also, it is necessary to remember that C strings are null-terminated and to take special action in either the MACRO routine or the C routine to allow for this.

Example 3.8, "VAX MACRO Program Calling a VSI C Function" shows a MACRO routine that calls a C routine with three arguments, passing one by value, one by reference, and one by descriptor. It is followed by the source for the called C routine.

Example 3.8. VAX MACRO Program Calling a VSI C Function

```
;------------------------------------------------------------------------
; Beginning of MACRO program
;------------------------------------------------------------------------
        .extrn  dbroutine        ; The C routine
;------------------------------------------------------------------------
; Local Data
;------------------------------------------------------------------------

        .psect        data        rd,nowrt,noexe

        ft$$t_part_num:        .ascii /WidgitGadget/
        ft$$t_query_mode:        .ascii /I/
        ft$$s_query_mode =        <. - ft$$t_query_mode>
        ft$$l_protocol_buff:        .blkl        1
        ft$$kd_part_num_dsc:
                        .word   12
                        .word        0
                        .address ft$$t_part_num


;------------------------------------------------------------------------
; Entry Point
;------------------------------------------------------------------------
        .psect        ft_code        rd,nowrt,exe
        .entry dbtest        ^m<r2,r3,r4,r5,r6,r7,r8>

;+
; call C routine for data base lookup
;-
        movl        #1,r3
        pushal      ft$$kd_part_num_dsc             ; Descriptor for part
 number
        pushal      ft$$t_query_mode                ; Mode to call
        pushl       #1                              ; Status
        calls       #3, dbroutine                   ; Check the data base
99$:
        ret

        .end dbtest
;------------------------------------------------------------------------
; End of MACRO program
;------------------------------------------------------------------------
```

```
/*
 *  Beginning of VSI C code for dbroutine:
 */

#include <stdio.h>
#include <descrip.h>
#include <stdlib.h>
#include <string.h>

/* Structure pn_desc is the format of the descriptor
   passed by the macro routine. */

extern     struct
       mydescript    {
                short  pn_len;
                short  pn_zero;
                char   *pn_addr;
                };


int  dbroutine (int status,                    /* Passed by value      */
                char *action,                  /* Passed by reference  */
                struct mydescript *name_dsc)   /* Passed by descriptor */
{
        char *part_name;

        /* Allocate space to put the null-padded name string. */
        part_name = malloc(name_dsc->pn_len + 1);
        memcpy( part_name,name_dsc -> pn_addr ,name_dsc -> pn_len);

        /* Remember that C array bounds start at 0 */
        part_name[name_dsc -> pn_len] = '\0';


        printf (" Status is %d\n", status);
        printf (" Length  is %d\n",name_dsc -> pn_len);
        printf (" Part_name is %s\n",part_name);
        printf (" Request is %c\n",*action);
        status = 1;
        return(status);
}  /*  End of VSI C code for dbroutine  */
```

Example 3.8, "VAX MACRO Program Calling a VSI C Function" produces the following output:

```
 Status is 1
 Length  is 12
 Part_name is WidgitGadget
 Request is I
```

Example 3.9, "VSI C Program Calling a VAX MACRO Program" shows a VSI C program that calls a VAX MACRO program.

Example 3.9. VSI C Program Calling a VAX MACRO Program

```
/* Beginning of VSI C function  */

#include <stdio.h>
```

```
#include <descrip.h>

int zapit( int status, int *action, struct dsc$descriptor_s *descript);


main(void)
{
    int status=255, argh = 99;
    int *action = &argh;
    $DESCRIPTOR(name_dsc,"SuperEconomySize");

    printf(" Before calling ZAPIT: \n");
    printf(" Status was %d \n",status);
    printf(" Action contained %d and *action contained %d \n" ,action,
             *action);
    printf(" And the thing described by the descriptor was %s\n",
             name_dsc.dsc$a_pointer);


    if (zapit(status,action,&name_dsc) && 1)
    {
      printf(" Ack, the world has been zapped! \n");
      printf(" Status is %d \n",status);
      printf(" Action contains %d and *action contains %d \n" ,action,
               *action);
      printf(" And the address of the thing described by the descriptor is
 %d\n",
              name_dsc.dsc$a_pointer);

    }
}  /*  End of VSI C function */


;------------------------------------------------------------------------
;  Beginning of VAX MACRO source code for zapit
;------------------------------------------------------------------------
; Entry Point
;------------------------------------------------------------------------
        .psect        ft_code        rd,nowrt,exe
        .entry zapit        ^m<r2,r3,r4,r5,r6,r7,r8>

;+
; Maliciously change parameters passed by the C routine.
;
; The first parameter is passed by value, the second by
; reference, and the third by descriptor.
;-

        movl        4(ap), @8(ap)        ;Change the by-reference parameter
                                         ;to the first parameter's value.

        movl        12(ap), r2
        movl        #0,4(r2)             ;Zap address of string in
                                         ;descriptor.

        ; Return -1 to signal successful destruction.
        movl        #-1,r0
        ret
```

```
        .end
;------------------------------------------------------------------------
;  End of VAX MACRO source code for zapit
;------------------------------------------------------------------------
```

Example 3.9, "VSI C Program Calling a VAX MACRO Program" produces the following output:

```
 Before calling ZAPIT:
 Status was 255
 Action contained 2146269556 and *action contained 99
 And the thing described by the descriptor was SuperEconomySize
 Ack, the world has been zapped!
 Status is 255
 Action contains 2146269556 and *action contains 255
 And the address of the thing described by the descriptor is 0
```

3.3.3. Calling VSI BASIC

Calling routines written in VSI BASIC from VSI C programs or vice versa is straightforward. By default, VSI BASIC passes arguments by reference, except for arrays and strings, which are passed by descriptor. In some cases, these defaults may be overridden by explicitly specifying the desired parameter-passing mechanisms in the VSI BASIC program. However, if an argument is a constant or an expression, the actual argument passed refers to a local copy of the specified argument's value.

Strings in VSI BASIC are not terminated by a null character, which is done by VSI C. As a result, passing strings between VSI BASIC and VSI C routines requires you to do additional work. You may choose to add an explicit null character to a VSI BASIC string before passing it to a VSI C routine, or you may prefer to code the VSI C routine to obtain the string's length from its descriptor.

Example 3.10, "VSI C Function Calling a VSI BASIC Function" shows a VSI C program that calls a VSI BASIC function with a variety of argument data types.

Example 3.10. VSI C Function Calling a VSI BASIC Function

```
/*
 *  Beginning of VSI C function:
 */

#include <stdio.h>
#include <descrip.h>

        extern        int        basfunc ();

        main(void)
        {
                int        i = 508;
                float        f = 649.0;
                double        d = 91.50;
                struct
                {
                        short        s;
                        float        f;
                }        s = { -2, -3.14 };
                $DESCRIPTOR (string1, "A C string");
```

```
        printf ("BASIC returned %d\n",
                basfunc (&i, &f, &d, &s, &string1, "bye"));
}  /*  End of VSI C function   */


!  Beginning of the BASIC program
FUNCTION INTEGER basfunc (INTEGER i, REAL f, DOUBLE d, x s, &
                          STRING string1,                   &
                          STRING string2 = 3 BY REF)

    RECORD        x
        WORD    s
        REAL    f
    END RECORD x


    PRINT 'i = '; i
    PRINT 'f = '; f
    PRINT 'd = '; d
    PRINT 's::s = '; s::s
    PRINT 's::f = '; s::f
    PRINT 'string1 = '; string1
    PRINT 'string2 = '; string2

END FUNCTION -15
! End of the BASIC program
```

Example 3.10, "VSI C Function Calling a VSI BASIC Function" produces the following output:

```
i =  508
f =  649
d =  91.5
s::s = -2
s::f = -3.14
string1 = A C string
string2 = bye
BASIC returned -15
```

Example 3.11, "VSI BASIC Program Calling a VSI C Function" shows a VSI BASIC program that calls a VSI C function.

Example 3.11. VSI BASIC Program Calling a VSI C Function

```
! Beginning of the BASIC program:
PROGRAM example

    EXTERNAL STRING FUNCTION cfunc (INTEGER BY VALUE, &
                                    INTEGER BY VALUE, &
                                    STRING BY DESC)

    s$ = cfunc (5, 3, "abcdefghi")
    PRINT "substring is "; s$

END PROGRAM
! End of the BASIC program


/*
```

```
 *  Beginning of VSI C function:
 */

#include <descrip.h>
int str$copy_dx();

        /*
         *  This routine simulates a BASIC function whose return
         *  value is a STRING.  It returns the substring that is `length'
         *  characters long, starting from the offset `offset' (0-based)
         *  in the input string described by the descriptor pointed to
         *  by `in_str'.
         */

        void        cfunc (struct dsc$descriptor_s *out_str,
                           int offset,
                           int length,
                           struct dsc$descriptor_s *in_str)
        {
                /* Declare a string descriptor for the substring. */
                struct        dsc$descriptor        temp;

                /* Check that the desired substring is wholly
                   within the input string. */
                if (offset + length > in_str -> dsc$w_length)
                        return;


                /* Fill in the substring descriptor. */
                temp.dsc$w_length = length;
                temp.dsc$a_pointer = in_str -> dsc$a_pointer + offset;
                temp.dsc$b_dtype = DSC$K_DTYPE_T;
                temp.dsc$b_class = DSC$K_CLASS_S;

                /* Copy the substring to the return string. */
                str$copy_dx (out_str, & temp);
        }  /*  End of VSI C function  */
```

Example 3.11, "VSI BASIC Program Calling a VSI C Function" produces the following output:

```
substring is fgh
```

3.3.4. Calling VSI Pascal

Like VSI Fortran and VSI BASIC, there are certain considerations that you must take into account when calling VSI Pascal from VSI C and vice versa. When calling VSI Pascal from VSI C, VSI Pascal expects all parameters to be passed by reference. In VSI Pascal, there are two different types of semantics: value and variable. The value semantics in VSI Pascal are different from passing by value in VSI C. Because they are different, you must specify the address of the C parameter.

VSI Pascal also expects all strings to be passed by descriptor. If you use the CLASS_S descriptor, the string is passed by using VSI Pascal semantics. If the content of the string is changed, it is not reflected back to the caller.

Example 3.12, "VSI C Function Calling a VSI Pascal Routine" is an example of how to call a VSI Pascal routine from VSI C.

Example 3.12. VSI C Function Calling a VSI Pascal Routine

```
/*
 * Beginning of VSI C function:
 */

#include <descrip.h>

/* This program demonstrates how to call a Pascal routine
   from a C function. */

/* A Pascal routine called by a C function. */
extern        void        Pascal_Routine ();

main()
{
    struct dsc$descriptor_s to_Pascal_by_desc;
    char *Message = "The_Max_Num";
    int to_Pascal_by_value = 100,
    to_Pascal_by_ref = 50;


    /* Construct the descriptor. */
    to_Pascal_by_desc.dsc$a_pointer = Message;
    to_Pascal_by_desc.dsc$w_length = strlen (Message);
    to_Pascal_by_desc.dsc$b_class = DSC$K_CLASS_S;
    to_Pascal_by_desc.dsc$b_dtype = DSC$K_DTYPE_T;

    /* Pascal expects a calling routine to pass parameters by reference. */


    Pascal_Routine(&to_Pascal_by_value, &to_Pascal_by_ref,
 &to_Pascal_by_desc);

    printf ("\nWhen returned from Pascal:\nto_Pascal_by_value is still \
%d\nBut to_Pascal_by_ref is %d\nand Message is still %s\n",
            to_Pascal_by_value, to_Pascal_by_ref,
            to_Pascal_by_desc.dsc$a_pointer);
}  /*  End of VSI C function   */

{
  Beginning of Pascal routine
}

MODULE C_PASCAL(OUTPUT);

 { This Pascal routine calls the Pascal MAX function
   to determine the maximum value between
   'from_c_by_value` and 'from_c_by_ref`, and then
   assigns the result back to 'from_c_by_ref`.
   It also tries to demonstrate the results of passing
   a by-descriptor mechanism.
   It is called from a C main function.
 }
[GLOBAL]PROCEDURE Pascal_Routine
      (          from_c_by_value :INTEGER;
      VAR from_c_by_ref :INTEGER;
          from_c_by_desc :[ CLASS_S ] PACKED ARRAY [l1..u1:INTEGER] OF CHAR
```

```
      );

    VAR
        today_is : PACKED ARRAY [1..11] OF CHAR;

    BEGIN


    { Display the contents of formal parameters. }
    WRITELN;
    WRITELN ('Parameters passed from C function: ');
    WRITELN ('from_c_by_value: ', from_c_by_value:4);
    WRITELN ('from_c_by_ref: ', from_c_by_ref:4);
    WRITELN ('from_c_by_desc: ', from_c_by_desc);

    { Assign the maximum value into 'from_c_by_ref` }
    from_c_by_ref := MAX (from_c_by_value, from_c_by_ref);

    { Change the content of 'from_Pascal_by_value` --
      to show that the value did not get
      reflected back to the caller.
     }
    from_c_by_value := 20;


    { Put the results of DATE into 'from_c_by_desc`
      to show that the CLASS_S is only valid with
      comformant strings passed by value.
     }
    DATE (today_is);
    from_c_by_desc := today_is;
    WRITELN ('**********************');
    WRITELN ('from_c_by_desc is changed to today''s date: "',
                      from_c_by_desc, '"');
    WRITELN ('BUT, this will not reflect back to the caller.');
END;
END.
{
  End of Pascal routine
}
```

Example 3.12, "VSI C Function Calling a VSI Pascal Routine" produces the following output:

```
from_c_by_value:  100
from_c_by_ref:   50
from_c_by_desc: The_Max_Num
**********************
from_c_by_desc is changed to today's date "26-MAY-1992"
BUT, this will not reflect back to the caller.

When returned from Pascal:
to_Pascal_by_value is still 100
to_Pascal_by_ref is  100
and Message is still The_Max_Num
```

There are also some considerations when calling VSI C from VSI Pascal. For example, you can use mechanism specifiers such as %IMMED, %REF, and %STDESCR in VSI Pascal. When you use the %IMMED mechanism specifier, the compiler passes a copy of a value rather than an address. When you

use the %REF mechanism specifier, the address of the actual parameter is passed to the called routine, which is then allowed to change the value of the corresponding actual parameter. When you use the %STDESCR mechanism specifier, the compiler generates a fixed-length descriptor of a character-string variable and passes its address to the called routine. For more information on these mechanism specifiers and others, see the VSI Pascal documentation.

Another consideration is that VSI Pascal does not null-pad strings. Therefore, you must add a null character to make the string a C string. Also, when passing a string from VSI Pascal to VSI C, you can declare a structure declaration in VSI C that corresponds to the VSI Pascal VARYING TYPE declaration.

Example 3.13, "VSI Pascal Program Calling a VSI C Function" shows an example of how to call VSI C from VSI Pascal.

Example 3.13. VSI Pascal Program Calling a VSI C Function

```
{
  Beginning of Pascal function:
}

PROGRAM PASCAL_C (OUTPUT);

CONST
        STRING_LENGTH = 80;

TYPE
        STRING = VARYING [STRING_LENGTH] OF CHAR;

VAR
    by_value : INTEGER;
    by_ref : STRING;
    by_desc: PACKED ARRAY [1..10] OF CHAR;


[EXTERNAL]
PROCEDURE DECC$CRTL_INIT; EXTERN;

[EXTERNAL]
PROCEDURE c_function
 (   %immed      by_value :  INTEGER;
     %ref        by_ref :    STRING ;
     %stdescr    by_desc:    PACKED ARRAY [l1..u1:INTEGER] OF CHAR
 );  EXTERN;


BEGIN

    { Establish the appropriate VSI C RTL environment for
       calling the VSI C RTL from Pascal.
    }
    DECC$CRTL_INIT;

    by_value := 1;

    {                NOTE
       Pascal does not null pad a string.
       Therefore, the LENGTH built-in function counts
```

```
       the null pad character while the VSI C library function strlen
       does not include the terminating null character.
     }

    by_ref := 'TO_C_BY_REF'(0)'';
    by_desc := 'TERM'(0)'';


    { Call a C function by passing parameters
       using foreign semantics.
    }
    c_function (by_value, by_ref, by_desc);

    WRITELN;
    WRITELN;
    WRITELN ('************************');
    WRITELN ('After calling C_FUNCTION: ');
    WRITELN;
    WRITELN ('by_value is still ',by_value:3);
    WRITELN ('however, by_ref contains ',by_ref,
                     ' (aka Your Terminal Type)');
    WRITELN ('and, by_desc still contains ',by_desc);

END.
{
  End of Pascal program
}

/*
 *  Beginning of VSI C function:
 *
 *
 * A C function called from the Pascal routine.
 * The parameters are passed to a C function
 * by value, by reference, and by descriptor,
 * respectively.
 */
#include <descrip.h>

/* A Pascal style of VARYING data type. */
struct Pascal_VARYING
{
    unsigned short        length;
    char                string[80];
};



/* This C function calls the VSI C RTL function getenv() and puts
 * your terminal type in 'from_Pascal_by_ref`.
 * It is called from a Pascal program.
 */
void        c_function  (unsigned char            from_Pascal_by_value,
                         struct Pascal_VARYING   *from_Pascal_by_ref,
                         struct dsc$descriptor_s *from_Pascal_by_desc
                         )
{
    char *term;
```

```
    /* Display the contents of formal parameters. */
    printf ("\nParameters passed from Pascal:\n");
    printf ("from_Pascal_by_value: %d\nfrom_Pascal_by_ref: %s\n\
from_Pascal_by_desc: %s\n", from_Pascal_by_value,
                            from_Pascal_by_ref -> string,
                            from_Pascal_by_desc -> dsc$a_pointer);

    if ((term = getenv(from_Pascal_by_desc -> dsc$a_pointer)) != 0)
    {
          /* Fill 'from_Pascal_by_ref` with new value. */
             strcpy (from_Pascal_by_ref -> string, term);
             from_Pascal_by_ref -> length = strlen (term);

          /* Change the contents of 'from_Pascal_by_value` --
           * to demonstrate that the value did not get
           * reflected back to the calling routine.
           */
             from_Pascal_by_value = from_Pascal_by_desc -> dsc$w_length
                                  + from_Pascal_by_ref -> length;
    }

    else
             printf ("\ngetenv\(\"TERM\"\) is undefined.");

}   /*  End of VSI C function   */
```

Example 3.13, "VSI Pascal Program Calling a VSI C Function" produces the following output:

```
Parameters passed from Pascal:
from_Pascal_by_value: 1
from_Pascal_by_ref: TO_C_BY_REF
from_Pascal_by_desc: TERM


************************
After calling C_FUNCTION:

by_value is still 1
however, by_ref contains vt200-80 (aka Your Terminal Type)
and, by_desc still contains TERM
```

3.4. Sharing Global Data

The following sections describe the methods involved in sharing VSI C program sections with data declared in other OpenVMS languages.

3.4.1. Sharing Program Sections with FORTRAN Common Blocks

In a FORTRAN program, separately compiled procedures can share data in declared common blocks, which specify the names of one or more variables to be placed in them. Each named common block represents a separate program section. Each procedure that declares the common block with the same name can access the same variable.

Example 3.14, "Sharing Data with a FORTRAN Program in Named Program Sections" shows a VSI C `extern` variable that corresponds to a FORTRAN common block with the same name.

Example 3.14. Sharing Data with a FORTRAN Program in Named Program Sections

```
C  FORTRAN program PRSTRING.FOR contains the following lines of code:

   SUBROUTINE PRSTRING
   CHARACTER*20 STRING
   COMMON /XYZ/ STRING

   TYPE 20, STRING
20 FORMAT (' ',A20)
   RETURN
   END

C  End of FORTRAN program


/*  VSI C program STRING.C contains the following lines of     *
 *  code:                                                      */

main(void)
{
  #pragma extern_model common_block  // Alpha only. On VAX systems, use
                                     // #pragma extern_model common_block
                                        shr
  extern char xyz[20];

  strncpy(xyz,"This is a string     ", sizeof xyz);
  prstring();
}
```

In *Example 3.14, "Sharing Data with a FORTRAN Program in Named Program Sections"*, the VSI C `extern` variable `xyz` corresponds to the FORTRAN common block named XYZ. The FORTRAN procedure displays the data in the block. When sharing program sections, both programs should declare corresponding variables to be of the same type.

Note the `#pragma extern_model common_block` preprocessor directive. This directive sets the model for external variables to the `common_block` model, which is the one used by `VAX C`. The default external model for VSI C is the `relaxed_refdef` model. For more information on the `#pragma extern_model common_block` preprocessor directive, see *Section 5.4.5, "#pragma extern_model Directive"*.

To share data in more than one variable in a program section with a FORTRAN program, the VSI C variables must be declared within a structure, as shown in *Example 3.15, "Sharing Data with a FORTRAN Program in a VSI C Structure"*.

Example 3.15. Sharing Data with a FORTRAN Program in a VSI C Structure

```
C  FORTRAN program FNUM.FOR contains the following lines of code:

   SUBROUTINE FNUM
   INTEGER*4 INUM,JNUM,KNUM
   COMMON /NUMBERS/ INUM,JNUM,KNUM
```

```
   TYPE 10, (INUM,JNUM,KNUM)
10 FORMAT (3I8)
   RETURN
   END

C  End of FORTRAN program


/*  VSI C program NUMBERS.C contains the following lines of    *
 *  code:                                                      */
struct xs
   {
      int first;
      int second;
      int third;
   };

#pragma extern_model common_block

main()
{
   extern  struct  xs  numbers;

   numbers.first = 1;
   numbers.second = 2;
   numbers.third = 3;
   fnum();
}
```

In *Example 3.15, "Sharing Data with a FORTRAN Program in a VSI C Structure"*, the `int` variables declared in the VSI C structure numbers correspond to the FORTRAN INTEGER*4 variables in the COMMON of the same name.

Also, note the `#pragma extern_model common_block` preprocessor directive. This directive sets the model for external variables to the `common_block` model, which is the one used by `VAX C`. The default external model for VSI C is the `relaxed_refdef` model. For more information on the `#pragma extern_model common_block` preprocessor directive, see *Section 5.4.5, "#pragma extern_model Directive"*.

3.4.2. Sharing Program Sections with PL/I Externals

A P/LI variable with the EXTERNAL attribute corresponds to a FORTRAN common block and to a VSI C `extern` variable in the `common_block` external model. Example *Example 3.16, "Sharing Data with a PL/I Program in Named Program Sections"* and Example *Example 3.17, "Sharing Data with a PL/I Program in a VSI C Structure"* show how a program section is shared between VSI C and P/LI.

A PL/I EXTERNAL CHARACTER attribute corresponds to a VSI C `extern char` variable, but PL/I character strings are not necessarily null-terminated. In *Example 3.16, "Sharing Data with a PL/I Program in Named Program Sections"*, VSI C and P/LI use the same variable to manipulate the character string that resides in a program section named XYZ.

Example 3.16. Sharing Data with a PL/I Program in Named Program Sections

```
/*  PL/I program PRSTRING.PLI contains the following lines of code:  */

PRSTRING: PROCEDURE;
```

```
   DECLARE XYZ EXTERNAL CHARACTER(20);

   PUT SKIP LIST(XYZ);
   RETURN;

END PRSTRING;

/*  End of PL/I program  */


/*  VSI C program STRING.C contains the following lines of     *
 *  code:                                                      */

main(void)
{
   extern char xyz[20];

   strncpy(xyz,"This is a string    ", sizeof xyz);
   prstring();
}
```

The PL/I procedure PRSTRING writes out the contents of the external variable `XYZ`.

PL/I also has a structure type similar (in its internal representation) to the `struct` keyword in VSI C. Moreover, P/LI can output aggregates, such as structures and arrays, in fairly simple stream-output statements; consider *Example 3.17, "Sharing Data with a PL/I Program in a VSI C Structure"*.

Example 3.17. Sharing Data with a PL/I Program in a VSI C Structure

```
/*  PL/I program FNUM.PLI contains the following lines of code:  */

FNUM:  PROCEDURE;
    /* EXTERNAL STRUCTURE CONTAINING THREE INTEGERS */
    DECLARE 1 NUMBERS EXTERNAL,
            2 FIRST FIXED(31),
            2 SECOND FIXED(31),
            2 THIRD FIXED(31);

    PUT SKIP LIST('Contents of structure:',NUMBERS);
    RETURN;
END FNUM;

/*  End of PL/I program   */


/*  VSI C program NUMBERS.C contains the following lines of    *
 *  code:                                                      */

struct xs
   {
      int first;
      int second;
      int third;
   };

main()
{
```

```
   extern  struct  xs  numbers;

   numbers.first = 1;
   numbers.second = 2;
   numbers.third = 3;
   fnum();
}
```

The PL/I procedure FNUM writes out the complete contents of the external structure NUMBERS; the structure members are written out in the order of their storage in memory, which is the same as for a VSI C structure.

3.4.3. Sharing Program Sections with MACRO Programs

In a MACRO program, the .PSECT directive sets up a separate program section that can store data or MACRO instructions. The attributes in the .PSECT directive describe the contents of the program section.

Example 3.18, "Sharing Data with a MACRO Program in a VSI C Structure" shows how to set up a psect in a MACRO program that allows data to be shared with a VSI C program.

Example 3.18. Sharing Data with a MACRO Program in a VSI C Structure

```
;  MACRO source file SET_VALUE.MAR contains the following lines of code:

           .entry  set_value,^M<>

           movl     #1,first
           movl     #2,second
           movl     #3,third
           ret

           .psect  example pic,usr,ovr,rel,gbl,noshr,-
                   noexe,rd,wrt,novec,long
first:     .blkl
second:    .blkl
third:     .blkl

           .end

;  End of MACRO source file


/*  VSI C program NUMBERS.C contains the following lines of    *
 *  code:                                                      */

#pragma extern_model common_block

struct xs
   {
      int first;
      int second;
      int third;
   }  example;
```

```
main()
{
   set_value();

   printf("example.first = %d\n",  example.first);
   printf("example.second = %d\n", example.second);
   printf("example.third = %d\n",  example.third);
}
```

The MACRO program initializes the locations first, second, and third in the psect named `example` and passes these values to the VSI C program. The locations are referenced in the VSI C program as members of the external structure named `example`.

Also, note the `#pragma extern_model common_block` preprocessor directive. This directive sets the model for external variables to the `common_block` model, which is the one used by VAX C. The default external model for VSI C is the `relaxed_refdef` model. For more information on the `#pragma extern_model common_block` preprocessor directive, see *Section 5.4.5, "#pragma extern_model Directive"*.

3.5. OpenVMS Run-Time Library Routines

The OpenVMS Run-Time Library (RTL) is a library of prewritten, commonly used routines that perform a wide variety of functions. These routines are grouped according to the types of tasks they perform, and each group has a prefix that identifies those routines as members of a particular OpenVMS RTL facility. *Table 3.10, "OpenVMS Run-Time Library Facilities"* lists all the language-independent, run-time library facility prefixes and the types of tasks each facility performs.

Table 3.10. OpenVMS Run-Time Library Facilities

Facility Prefix	Types of Tasks Performed
LIB$	Library routines that obtain records from devices, manipulate strings, convert data types for I/O, allocate resources, obtain system information, signal exceptions, establish condition handlers, enable detection of hardware exceptions, and process cross-reference data.
MTH$	Mathematics routines that perform arithmetic, algebraic, and trigonometric calculations.
OTS$	General-purpose routines that perform tasks such as data-type conversions as part of a compiler's generated code.
SMG$	Screen management routines that are used in designing, composing, and keeping track of complex images on a video screen.
STR$	String manipulation routines that perform such tasks as searching for substrings, concatenating strings, and prefixing and appending strings.

The OpenVMS run-time library routines are documented in detail in the following operating system documentation:

- *VSI OpenVMS RTL Library (*LIB$*) Manual*
- *OpenVMS VAX RTL Mathematics (*MTH$*) Manual*
- *Portable Mathematics Library*

- *VSI OpenVMS RTL General Purpose (*OTS$*) Manual*
- *VSI OpenVMS RTL Screen Management (*SMG$*) Manual*
- *VSI OpenVMS RTL String Manipulation (*STR$*) Manual*

3.6. OpenVMS System Services Routines

System services are prewritten system routines that perform a variety of tasks, such as controlling processes, communicating among processes, and coordinating I/O.

Unlike the OpenVMS Run-Time Library (RTL) routines, which are divided into groups by facility, all system services share the same facility prefix (SYS$). However, these services are logically divided into groups that perform similar tasks. *Table 3.11, "OpenVMS System Services"* describes these groups.

Table 3.11. OpenVMS System Services

Group	Types of Tasks Performed
AST	Allows processes to control the handling of asynchronous system traps (ASTs).
Change mode	Changes the access mode of particular routines.
Condition handling	Designates condition handlers for special purposes.
Event flag	Clears, sets, reads, and waits for event flags, and associates with event flag clusters.
Information	Returns information about the system, queues, jobs, processes, locks, and devices.
Input/Output	Performs I/O directly without going through RMS.
Lock management	Enables processes to coordinate access to shareable system resources.
Logical names	Provides methods of accessing and maintaining pairs of character-string logical names and equivalence names.
Memory management	Increases or decreases available virtual memory, controls paging and swapping, and creates and accesses shareable files of code or data.
Process control	Creates, deletes, and controls execution of processes.
Security	Enhances the security of OpenVMS systems.
Time and Timing	Schedules events and obtains and formats binary time values.

System services are documented in detail in the *VSI OpenVMS System Services Reference Manual*.

The routines that provide a programming interface to various OpenVMS utilities are described in the *VSI OpenVMS Utility Routines Manual*.

3.7. Calling Routines

The basic steps for calling routines are the same whether you are calling a routine written in VSI C, a routine written in some other OpenVMS language, a system service, or an OpenVMS Run-Time Library (RTL) routine. The following sections outline the procedures for calling non-VSI C routines.

3.7.1. Determining the Type of Call

Before calling an external routine, you must first determine whether the call should be a procedure call or a function call. Call a routine as a procedure if it does not return a value. Call a routine as a function if it returns any type of value.

3.7.2. Declaring an External Routine and Its Arguments

To call an external routine or system routine, you need to declare it as an external function and to declare the names, data types, and passing mechanisms of its arguments. Arguments can be either required or optional.

Include the following information in a routine declaration:

- The name of the external routine
- The data types of all the routine parameters (optional)
- The data type of the return value if it is a function
- The `void` keyword if it is a procedure

The following example shows how to declare an external routine and its arguments:

```
char func_name (int x, char y);
```

Header files are available to declare commonly used external routines. Using them will save you a lot of work. See Sections *Section 1.3.1.1, "Including Header Files"* and *Section 1.3.1.2, "Listing Header Files"* in this manual for information on listing and including header files.

3.7.3. Calling the External Routine

After declaring an external routine, you can invoke it. To invoke a function, you must specify the name of the routine being invoked and all arguments required for that routine. Make sure the data types for the actual arguments you are passing coincide with those of the parameters you declared earlier, and with those declared in the routine. The following example shows how to invoke the function declared in *Section 3.7.2, "Declaring an External Routine and Its Arguments"*:

```
ret_status = func_name(1,'a');
```

3.7.4. System Routine Arguments

All system routine arguments are described in terms of the following information:

- OpenVMS usage
- Data type
- Type of access allowed
- Passing mechanism

OpenVMS usages are data structures that are layered on the standard OpenVMS data types. For example, the OpenVMS usage mask_longword signifies an unsigned longword integer that is used as a bit mask, and the OpenVMS usage floating_point represents any OpenVMS floating-point data type. *Table 3.12, "VSI C Implementation"* lists all the OpenVMS usages and the VSI C types you need to implement them.

Table 3.12. VSI C Implementation

OpenVMS Data Type	VSI C Declaration
access_bit_names	user-defined[1]
access_mode	unsigned char
address	int *pointer [24]
address_range	int *array [2] [234]
arg_list	user-defined[1]
ast_procedure	pointer to a function[2]
boolean	unsigned long int
byte_signed	char
byte_unsigned	unsigned char
channel	unsigned short int
char_string	char array[n][35]
complex_number	user-defined[1]
cond_value	unsigned long int
context	unsigned long int
date_time	user-defined[1]
device_name	char array[n][35]
ef_cluster_name	char array[n][35]
ef_number	unsigned long int
exit_handler_block	user-defined[1]
fab	#include fab from text library struct FAB
file_protection	unsigned short int, or user-defined[1]
floating_point	float or double
function_code	unsigned long int or user-defined[1]
identifier	int *pointer[24]
io_status_block	user-defined[1]
item_list_2	user-defined[1]
item_list_3	user-defined[1]
item_list_pair	user-defined[1]
item_quota_list	user-defined[1]
lock_id	unsigned long int

OpenVMS Data Type	**VSI C Declaration**
lock_status_block	user-defined[1]
lock_value_block	user-defined[1]
logical_name	char array[n][35]
longword_signed	long int
longword_unsigned	unsigned long int
mask_byte	unsigned char
mask_longword	unsigned long int
mask_quadword	user-defined[1]
mask_word	unsigned short int
null_arg	unsigned long int
octaword_signed	user-defined[1]
octaword_unsigned	user-defined[1]
page_protection	unsigned long int
procedure	pointer to function[2]
process_id	unsigned long int
process_name	char array[n][35]
quadword_signed	user-defined[1]
quadword_unsigned	user-defined[1]
rights_holder	user-defined[1]
rights_id	unsigned long int
rab	#include rab struct RAB
section_id	user-defined[1]
section_name	char array[n][35]
system_access_id	user-defined[1]
time_name	char array[n][35]
uic	unsigned long int
user_arg	user-defined[1]
varying_arg	user-defined[1]
vector_byte_signed	char array[n][35]
vector_byte_unsigned	unsigned char array[n][35]
vector_longword_signed	long int array[n][35]
vector_longword_unsigned	unsigned long int array[n][35]
vector_quadword_signed	user-defined[1]

OpenVMS Data Type	VSI C Declaration
vector_quadword_unsigned	user-defined[1]
vector_word_signed	short int array[n][3][5]
vector_word_unsigned	unsigned short int array[n][3][5]
word_signed	short int
word_unsigned	unsigned short int

[1]The declaration of a user-defined data structure depends on how the data will be used. Such data structures can be declared in a variety of ways, each of which is more suitable to specific applications.

[2]The term pointer refers to several declarations involving pointers. Pointers are declared with special syntax and are associated with the data type of the object being pointed to. This object is often user-defined.

[4]The data type specified can be changed to any valid VSI C data type.

[3]The term array denotes the syntax of a VSI C array declaration.

[5]The size of the array must be substituted for n.

If a system routine argument is optional, it will be indicated in the format section of the routine description in one of two ways, as follows:

- [,optional-argument]
- ,[optional-argument]

If the comma appears outside the brackets, you must pass a 0 by value to indicate the place of the omitted argument. If the comma appears inside the brackets, you can omit the argument if it is the last argument in the list.

For more information, see the *OpenVMS Programming Interfaces: Calling a System Routine* manual. This manual describes the OpenVMS programming interface and defines the standard conventions to call an OpenVMS system routine from a user procedure. The Alpha and VAX data type implementations for various high-level languages are also presented.

3.7.5. Symbol Definitions

Many system routines depend on values that are defined in separate symbol definition files. OpenVMS RTL routines require you to include symbol definitions when you are calling a Screen Management facility routine or a routine that is a jacket to a system service. A *jacket* routine provides an interface to the corresponding system service. For example, the routine LIB$SYS_ASCTIM is a jacket routine for the $ASCTIM system service.

If you are calling a system service, you must include the `<ssdef.h>` header file to check the status. Many system services require other symbol definitions as well. To determine whether you need to include other symbol definitions for the system service you want to use, see the documentation for that particular system service. If the documentation states that values are defined in a macro, you must include those symbol definitions in your program.

For example, the description for the flags parameter in the SYS$MGBLSC (Map Global Section) system service states that “Symbolic names for the flag bits are defined by the $SECDEF macro.” Therefore, when you call SYS$MGBLSC you must include the definitions provided in the $SECDEF macro by including the `<secdef.h>` header file.

In VSI C, a header file is included as follows:

```
#include <ssdef.h>
```

To obtain a list of all VSI C header files, see *Section 1.3.1.2, "Listing Header Files"*.

3.7.6. Condition Values

Many system routines return a condition value that indicates success or failure; this value can be either returned or signaled. If a condition value is returned, then you must check the returned value to determine whether the call to the system routine was successful. Otherwise, the condition value is signaled to your program instead of being written to a storage location.

Condition values indicating success appear first in the list of condition values for a particular routine, and success codes have odd values. A success code that is common to many system routines is the condition value SS$_NORMAL, which indicates that the routine completed normally and successfully. If the condition value is returned, then you can test for SS$_NORMAL as follows:

```
      if (ret_status != SS$_NORMAL)
             LIB$STOP();
```

Because all success codes have odd values, you can check a return status for any success code. For example, you can cause execution to continue only if a success code is returned by including the following statements in your program:

```
if ((ret_status & 1) != 0)
     LIB$STOP (ret_status);
```

In general, you can check a return status for a particular success or failure code or you can test the condition value returned against all success codes or all failure codes.

3.7.7. Checking System Service Return Values

It is customary in OpenVMS programming to compare the return status of a system service with a global symbol, not with the literal value associated with a particular return status. Consequently, a high-level language program should define the possible return status values for a service as symbolic constants. In VSI C, you can do this by including the `<ssdef.h>` header file; *Example 3.19, "Checking System Service Return Values"* shows how this is done.

Example 3.19. Checking System Service Return Values

```
/*  This program shows how to compare the status of a system   *
 *  service with a global symbol.                              */
#include <stdlib.h>
                                    /*  Define system service   *
                                     *   status values          */
#include <ssdef.h>
#include <stdio.h>

                                    /*  Declaration of the      *
                                     *   service (not required) */
int  SYS$SETEF();

int main(void)
{
                                    /*  To hold the status of   *
                                     *   SYS$SETEF              */
   int  efstatus;
                                    /*  Argument values for     *
```

```
                                    *   SYS$SETEF              */
   enum  cluster0
      {
         completion, breakdown, beginning
      }  event;
        .
        .
        .
   event =  completion;

                                   /*  Set the event flag      */
   efstatus =  SYS$SETEF(event);

                                   /*  Test the return status  */
   if (efstatus == SS$_WASSET)
      fprintf (stderr,"Flag was already set\n");
   else
      if (efstatus == SS$_WASCLR)
         fprintf(stderr, "Flag was previously clear\n");
      else
         fprintf(stderr,
                 "Could not set completion event flag.\n \
Possible programming error.\n");

   exit(efstatus);
}
```

The system service return status values (SS$_WASSET and SS$_WASCLR) in Example *Example 3.19, "Checking System Service Return Values"* are defined by the `<ssdef.h>` header file.

Error handling in *Example 3.19, "Checking System Service Return Values"* is typical of programs running on OpenVMS systems. Using the following statements, the example program attempts to provide a program-specific error message and then passes the offending error status to the caller:

```
else
   fprintf(stderr,
           "Could not set completion event flag.\n \
Possible programming error.\n");

exit(efstatus);
```

If you execute the program with DCL, it interprets any status value the program returns. DCL prints a standard error message on the terminal to provide you with more information about the failure. For example, if the program encounters the SS$_ILLEFC return status, DCL displays the following messages:

```
Could not set completion event flag.
Possible programming error.
%SYSTEM-F-ILLEFC, illegal event flag cluster.
```

3.8. Variable-Length Argument Lists in System Services

Most system services and other external procedures require a specific number of arguments, but some accept a variable number of optional arguments. Because VSI C function declarations do not show the

number of parameters expected by external functions unless a function prototype is used, the way you call an external function from a VSI C program depends on the semantics of the called function. You must supply the number of arguments that the external function expects. The rules are as follows:

- When optional arguments occur between required arguments, they cannot be omitted. If omitting such an argument is necessary—for example, to select a default action—the argument must be written as a zero.
- When optional arguments occur at the end of an argument list, the format of the function reference depends on the action of the called function as follows:
  - If the called function checks the number of arguments passed, you can omit optional trailing arguments from the function reference. System services generally do not check the length of the argument list.
  - If the called function does not check the number of arguments passed, all arguments must be present in the function reference.

For example, the function STR$CONCAT, in the Common Run-Time Library, concatenates from 2 to 254 strings into a single string. It has the following call format:

```
ret = STR$CONCAT(dst, src1,
src2[, src3,…src254]);
```

For more information about the STR$CONCAT function, see the *VMS Run-Time Library Routines Volume*.

The identifier *dst* is the destination for the concatenated string, and *src1*, *src2*, … *src254* are the source strings. All arguments are passed by descriptor. All but the first two source strings are optional. The function checks to see how many arguments are present in the call; if fewer than three (the destination and two sources) are present, the function returns an error status value. *Example 3.20, "Using Variable-Length Argument Lists"* shows a call to the STR$CONCAT function from VSI C.

Example 3.20. Using Variable-Length Argument Lists

```
/*  This example shows a call to STR$CONCAT.                    */
#include <stdlib.h>
#include <stdio.h>
#include <descrip.h>
#include <ssdef.h>

int STR$CONCAT();

int main(void)
{
   int ret;                          /*  Return status of        *
                                      *   STR$CONCAT             */

                                     /*  Destination array of    *
                                      *  concatenated strings    */
   char dest[21];

                                     /*  Create compile-time     *
                                      *   descriptors:           */
          $DESCRIPTOR(dst, dest);
   static $DESCRIPTOR(src1, "abcdefghij");
```

```
   static $DESCRIPTOR(src2, "klmnopqrst");

                                    /*  Concatenate strings     */
   ret =  STR$CONCAT(&dst, &src1, &src2);

                                    /* Test return status value */
   if (ret != SS$_NORMAL)
      fprintf(stderr,"Failed to concatenate strings.\n"),
      exit(ret);

                                    /*  Process string          */
   else
        dest[20] = '\0',
        printf("Resultant string: %s\n",dest);
}
```

3.9. Return Status Values

The status values from OpenVMS system service procedures are returned in general register R0. This return status value indicates the success or failure of the operation performed by the called procedure. In VSI C, passing a return status value in R0 is equivalent to a function returning `int`.

To obtain a return status value from any system procedure, declare the procedure as a function, as shown in the following example:

```
int  SYS$SETEF();
```

After declaring a procedure in this way, you can invoke the procedure as a function and obtain a return status value. In VSI C, such a declaration is needed only as program documentation; SYS$SETEF can be called without explicit declaration and will be interpreted by default as a function returning `int`.

This section describes the following topics:

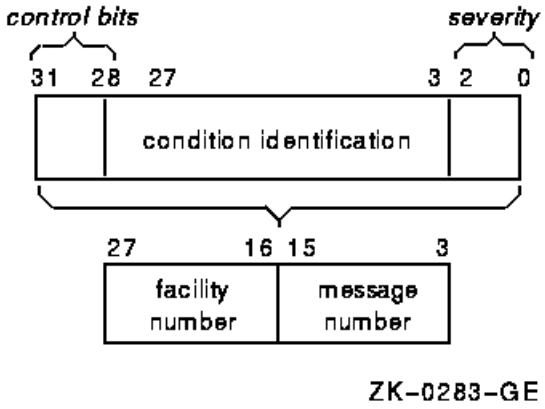
- The format of a return status value, that is, the meaning of particular bits within the value
- The way to manipulate return status values
- The recommended techniques for testing a return status value for success or failure or for a specific condition

3.9.1. Format of Return Status Values

All OpenVMS system procedures and programs use a longword value to communicate return status information. When a VSI C main function executing under the control of the DCL interpreter executes a `return` statement to return control to the command level, the command interpreter uses the return status value to conditionally display a message on the current output device.

To provide a unique means of identifying every return condition in the system, bit fields within the value are defined as shown in *Figure 3.5, "Bit Fields Within a Return Status Value"*.

Figure 3.5. Bit Fields Within a Return Status Value



The following list describes the division of this bit field:

control bits (31-28)

Define special action(s) to be taken. At present, only bit 28 is used. When set, it inhibits the printing of the message associated with the return status value at image exit. Bits 29 through 31 are reserved for future use by VSI and must `be 0`.

facility number (27-16)

A unique value assigned to the system component, or facility, that is returning the status value. Within this field, bit 27 has a special significance. If bit 27 is clear, the facility is a VSI facility: the remaining value in the facility number field is a number assigned by the operating system. If bit 27 is set, the number indicates a customer-defined facility.

message number (15-3)

An identification number that specifically describes the return status or condition. Within this field, bit 15 has a special significance. If bit 15 is set, the message number is unique to the facility issuing the message. If bit 15 is clear, the message is issued by more than one system facility.

severity (2-0)

A numeric value indicating the severity of the return status. *Table 3.13, "Possible Severity Values"* shows the possible values in these three bits, and their meanings.

Table 3.13. Possible Severity Values

Value	Meaning
0	Warning
1	Success
2	Error
3	Informational
4	Severe error, FATAL
5-7	Reserved

Odd values indicate success (an informational condition is considered a successful status) and even values indicate failures (a warning is considered an unsuccessful status).

The following names are associated with these fields:

control bits bit 28 (inhibit message)	CONTROLINHIB_MSG
facility number bit 27 (customer facility)	FAC_NOCUST_DEF
message number bit 15 (facility specific)	MSG_NOFAC_SP
severity bit 0 (success)	SEVERITYSUCCESS

When testing return values in a VSI C program, either you can test only for successful completion of a procedure or you can test for specific return status values.

3.9.2. Manipulating Return Status Values

You can construct a structure or union that describes a return status value, but this method of manipulating return status values is not recommended. A status value is usually constructed or checked using bitwise operators. VSI C provides the `<stsdef.h>` header file, which contains preprocessor definitions to make this job easier. All the preprocessor symbols are named according to the following OpenVMS naming convention:

```
STS$type_name
```

STS

Identifies standard return status values.

type

One of the following characters denoting the type of the constant:

K	Represents a constant value
M	Represents a bit mask
S	Represents the bit size of a field
V	Defines the bit offset to the field

name

An abbreviation for the field name.

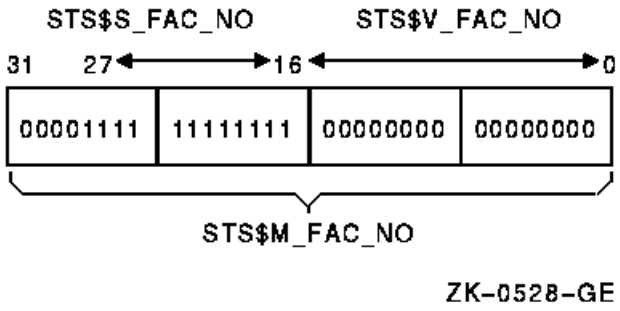
For example, the following constants are defined in `<stsdef.h>` for the facility number field, FAC_NO, which spans bits 16 through 27:

```
                                    /*  Size of field in bits   */
#define STS$S_FAC_NO  12

                                    /*  Bit offset to the       *
```

```
                                     *  beginning of the field  */
#define STS$V_FAC_NO   16

                                    /*  Bit mask of the field   */
#define STS$M_FAC_NO   0xFFF0000
```

Figure 3.6, "Internal Representation of a Status Value" shows how the status value is represented internally.

Figure 3.6. Internal Representation of a Status Value



Use the following expression to extract the facility number from a particular status value contained in the variable named status:

```
(status & STS$M_FAC_NO) >> STS$V_FAC_NO
```

In the previous example, the parentheses are required for the expression to be evaluated properly; the relative precedence of the bitwise AND operator (&) is lower than the precedence of the binary shift operator (>>).

3.9.3. Testing for Success or Failure

To test a return status value for success or failure, you need only test the success bit. A value of true in this bit indicates that the return value is a successful value.

Example 3.21, "Testing for Success" shows a program that checks the success bit.

Example 3.21. Testing for Success

```
/*  This program shows how to test the success bit.            */

#include <stdio.h>
#include <descrip.h>
#include <stsdef.h>
#include <starlet.h>
#include <stdlib.h>

int main(void)
{
   int  status;
   $DESCRIPTOR(name, "student");

   status = sys$setprn(&name);
```

```
   if (status & STS$M_SUCCESS)
                                    /*  Success code            */
        fprintf(stderr, "Successful completion");

   else
                                    /*  Failure code            */
      fprintf(stderr, "Failed to set process name.\n");
   exit(status);
}
```

The failure code in *Example 3.21, "Testing for Success"* causes the printing of a program-specific message indicating the condition that caused the program to terminate. The error status is passed to the DCL by the `exit` function, which then interprets the status value.

3.9.4. Testing for Specific Return Status Values

Each numeric return status value defined by the system has a symbolic name associated with it. The names of these values are defined as system global symbols, and you can access their values by referring to their symbolic names.

The global symbol names for OpenVMS return status values have the following format:

```
facility$_code
```

facility

An abbreviation or acronym for the system facility that defined the global symbol.

code

A mnemonic for the specific status value.

Table 3.14, "Facility Codes" shows some examples of facility codes used in global symbol names.

Table 3.14. Facility Codes

Facility	Description
SS	System services; these status codes are listed in the *VSI OpenVMS System Services Reference Manual*.
RMS	File system procedures; these status codes are listed in the *VSI OpenVMS Record Management Services Reference Manual*.
SOR	SORT procedures; these status codes are listed in the *VMS Sort/Merge Utility Manual*.

The definitions of the global symbol names for the facilities listed are located in the default VSI C object module libraries, so they are automatically located when you link a VSI C program that references them.

When you write a VSI C program that calls system procedures and you want to test for specific return status values using the symbol names, you must perform the following tasks:

1. Determine, from the documentation of the procedure, the status values that can be returned, and choose the values for which you want to provide specific tests.

2. Declare the symbolic name for each value of interest. The `<ssdef.h>` and `<rmsdef.h>` header files define the system service and RMS return status values, respectively. If you are checking return

status values from other facilities, such as the SORT utility, you must explicitly declare the return values as `globalvalue int`. Consider the following example:

```
globalvalue  int  SOR$_OPENIN;
```

3. Reference the symbols in your program.

Example 3.22, "Testing for Specific Return Status Values" shows a program that checks for specific return status values defined in the `<ssdef.h>` header file.

Example 3.22. Testing for Specific Return Status Values

```
/*  This program checks for specific return status values.     */
#include <stdlib.h>

#include <ssdef.h>
#include <stdio.h>
#include <descrip.h>

$DESCRIPTOR(message,"\07**Lunch_time**\07");

int main(void)
{

   int status =  SYS$BRDCST(&message,0);

   if (status != SS$_NORMAL)
      {
         if (status == SS$_NOPRIV)
            fprintf(stderr, "Can't broadcast; requires OPER \
privilege.");

         else
            fprintf(stderr, "Can't broadcast; some fatal \
error.");

         exit(status);
      }
}
```

3.10. Examples of Calling System Routines

This section provides complete examples of calling system routines from VSI C. *Example 3.23, "Passing Arguments to System Services"* shows the three mechanisms for passing arguments to system services and also shows how to test for status return codes. *Example 3.24, "Determining* $QIO *Completion"* shows various ways of testing for successful $QIO completion. *Example 3.25, "Using Time Routines"* shows how to use time conversion and set timer routines.

In addition to the examples provided here, the *VMS Run-Time Library Routines Volume* and the *VSI OpenVMS System Services Reference Manual* also provide examples for selected routines. See these manuals for help on using a specific system routine.

Example 3.23. Passing Arguments to System Services

```
/* GETMSG.C
```

```
   This program is an example showing the three mechanisms
   for passing arguments to system services.  It also
   shows how to test for specific status return
   codes from a system service call. */


#include <stdio.h>
#include <descrip.h>
#include <ssdef.h>
#include <lib$routines.h>


int main(void)
{
int message_id;
short message_len;
char text[133];
$DESCRIPTOR(message_text, text);
register status;


while (printf("\nEnter a message number <Ctrl/Z to quit>: "),
       scanf("%d", &message_id) != EOF)
   {
   /* Retrieve message associated with the number. */
   status = SYS$GETMSG(message_id, &message_len,
                       &message_text, 15, 0);


   /* Check for status conditions. */
   if (status == SS$_NORMAL)
      printf("\n%.*s\n", message_len, text);
   else if (status == SS$_BUFFEROVF)
      printf("\nBUFFER OVERFLOW - Text is: %.*s\n",
              message_len, text);
   else if (status == SS$_MSGNOTFND)
      printf("\nMESSAGE NOT FOUND.\n");
   else
      {
      printf("\nUnexpected error in $GETMSG call.\n");
      LIB$STOP(status);
      }
   }
}
```

Example 3.24. Determining $QIO Completion

```
/* ASYNCH.C
   This program shows various ways to determine
   $QIO completion. It also shows the use of an
   IOSB to obtain information about the I/O operation. */

#include <iodef.h>
#include <ssdef.h>
#include <descrip.h>
#include <lib$routines.h>
#include <stdio.h>
#include <starlet.h>
```

```
#include <string.h>

typedef struct
          {
          short cond_value;
          short count;
          int info;
          } io_statblk;

main(void)
{
char text_string[]  =  "This was written by the $QIO.";
register status;
short chan;
io_statblk status_block;
int AST_PROC();
$DESCRIPTOR (terminal, "SYS$COMMAND");

/*  Assign I/O channel.  */
if (((status = SYS$ASSIGN (&terminal, &chan,0,0)) & 1) != 1)
   LIB$STOP (status);

/*  Queue the I/O.  */
if (((status = SYS$QIO (1, chan, IO$_WRITEVBLK, &status_block,
          AST_PROC, &status_block, text_string,
          strlen(text_string),0,32,0,0)) & 1) != 1)
   LIB$STOP (status);

/*  Wait for the I/O operation to complete.  */
if (((status = SYS$SYNCH (1, &status_block)) & 1) != 1)
   LIB$STOP (status);
if ((status_block.cond_value &1) != 1)
   LIB$STOP(status_block.cond_value);

printf ("\nThe I/O operation and AST procedure are done.");
}

AST_PROC (*write_status)
io_statblk *write_status;

/* This function is called as an AST procedure. It uses
   the AST parameter passed to it by $QIO to determine
   how many characters were written to the terminal. */

{
printf("\nNumber of characters output is  %d", write_status->count);
printf("\nI/O completion status is %d", write_status->cond_value);
}
```

Example 3.25. Using Time Routines

```
/*  ALARM.C
    This program shows the use of time conversion
    and set timer routines.  */

#include <stdio.h>
#include <descrip.h>
#include <ssdef.h>
```

```
#include <lib$routines.h>
#include <starlet.h>

main(void)
{
#define event_flag 2
#define timer_id 3

typedef int quadword[2];

quadword delay_int;
$DESCRIPTOR(offset, "0 ::15.00");
char cur_time[24];
$DESCRIPTOR(cur_time_desc, cur_time);
int i;
unsigned state;
register status;

/* Convert offset from ASCII to binary format. */
if (((status=SYS$BINTIM(&offset, delay_int)) &1) != 1)
   LIB$STOP(status);

/* Output current time. */
if (((status=LIB$DATE_TIME(&cur_time_desc)) &1) != 1)
   LIB$STOP(status);
cur_time[23] = '\0';
printf("The current time is : %s\n", cur_time);

/* Set the timer to expire in 15 seconds. */
if (((status=SYS$SETIMR(event_flag, &delay_int,
                        0, timer_id)) &1) != 1)
   LIB$STOP(status);

/* Count to 1000000. */
printf("beginning count….\n");
for (i=0; i<=1000000; i++)
   ;

/* Check if the timer expired. */
switch (status = SYS$READEF(event_flag, &state))
 {
 case SS$_WASCLR : /* Cancel timer */
                   if (((status=SYS$CANTIM(timer_id, 0)) &1) != 1)
                       LIB$STOP(status);
                   printf("Count completed before timer expired.\n");
                   printf("Timer canceled.\n");
                   break;
 case SS$_WASSET : printf("Timer expired before count completed.\n");
                   break;
 default         : LIB$STOP(status);
                   break;
 }
}
```

Chapter 4. Data Storage and Representation

This chapter presents the following topics concerning VSI C data storage and representation on OpenVMS systems:

- Storage allocation (*Section 4.1, "Storage Allocation"*)
- Standard-conforming method of controlling external objects (*Section 4.2, "Standard-Conforming Method of Controlling External Objects"*)
- Global storage classes (*Section 4.3, "Global Storage Classes"*)
- Storage-class modifiers (*Section 4.4, "Storage-Class Modifiers"*)
- Floating-point numbers (*Section 4.5, "Floating-Point Numbers (float, double, long double)"*)
- Pointer conversions (*Section 4.6, "Pointer Conversions"*)
- Structure alignment (*Section 4.7, "Structure Alignment"*)
- Program sections (*Section 4.8, "Program Sections"*)

4.1. Storage Allocation

When you define a VSI C variable, the storage class determines not only its scope but also its location and lifetime. The lifetime of a variable is the length of time for which storage is allocated. For OpenVMS systems, storage for a VSI C variable can be allocated in the following locations:

- On the run-time stack
- In a machine register
- In a program section (psect)

Variables that are placed on the stack or in a register are temporary. For example, variables of the `auto` and `register` storage classes are temporary. Their lifetimes are limited to the execution of a single block or function. All declarations of the internal storage classes (`auto` and `register`) are also definitions; the compiler generates code to establish storage at this point in the program.

Program sections, or psects, are used for permanent variables; the lifetime of identifiers extends through the course of the entire program. A psect represents an area of virtual memory that has a name, a size, and a series of attributes that describe the intended or permitted usage of that portion of memory. For example, the compiler places variables of the static, external, and global storage classes in psects; you have some control as to which psects contain which identifiers. All declarations of the static storage class are also definitions; the compiler creates the psect at that point in the program. In VSI C, the first declaration of the external storage class is also a definition; the linker initializes the psect at that point in the program.

Note

The compiler does not necessarily allocate distinct variables to memory locations according to the order of appearance in the source code. Furthermore, the order of allocation can change as a result of

seemingly unrelated changes to the source code, command-line options, or from one version of the compiler to the next; it is essentially unpredictable. The only way to control the placement of variables relative to each other is to make them members of the same `struct` type or, on OpenVMS Alpha and IA-64 systems, by using the `noreorder` attribute on a named `#pragma extern_model strict_refdef`.

Table 4.1, "Location, Lifetime, and the Storage-Class Keywords" shows the location and lifetime of a variable when you use each of the storage-class keywords.

Table 4.1. Location, Lifetime, and the Storage-Class Keywords

Storage Class	Location	Lifetime
(Internal null)	Stack or register	Temporary
`[auto]`	Stack or register	Temporary
`register`	Stack or register	Temporary
`static`	Psect	Permanent
`extern`	Psect	Permanent
`globaldef`[1]	Psect	Permanent
`globalref`[1]	Psect	Permanent
`globalvalue`[1]	No storage allocated	Permanent

[1]The `globaldef`, `globalref`, and `globalvalue` storage-class specifiers are available only when compiling in VAX C compatibility mode.

For a comparison between the global and external storage classes, see *Section 4.3.2, "Comparing the Global and the External Storage Classes"*.

For more information about psects, see *Section 4.8, "Program Sections"*.

4.2. Standard-Conforming Method of Controlling External Objects

Sections *Section 4.3, "Global Storage Classes"* and *Section 4.4, "Storage-Class Modifiers"* describe the following external linkage storage-class specifiers and modifiers that are specific to VSI C for OpenVMS systems:

```
globaldef
globalref
globalvalue
noshare
readonly
_align
```

These keywords are supported by the VSI C compiler for compatibility purposes, and are available only in VAX C mode (/STANDARD=VAXC) and relaxed mode (/STANDARD=RELAXED).

However, the VSI C compiler also provides an alternative, standard-conforming method of controlling objects that have external linkage. To take advantage of this method, use the `#pragma extern_model` preprocessor directive and the /EXTERN_MODEL and /[NO]SHARE_GLOBALS command-line qualifiers.

The pragma and command-line qualifiers replace the VAX C mode storage-class specifiers (`globaldef`, `globalref`, `globalvalue`) and storage-class modifiers (`noshare` and `readonly`). They allow you to select the implementation model of external data and control the psect usage of your programs. The `_align` storage-class modifier is still used to ensure object alignment.

The pragma and command-line qualifier approach also has these advantages:

- Since the VAX C mode keywords do not follow standard C spelling rules, they cannot be provided in strict ANSI C mode. The pragma and qualifiers, however, can be used in any mode of the VSI C compiler.
- The pragma and qualifiers allow `extern` on OpenVMS systems to function in a manner more similar to other systems.
- The pragma and qualifiers make it easier for you to write OpenVMS shareable images with VSI C. Previously, that task required you to add an additional keyword to every declaration of external data.

For a description of the `#pragma extern_model` preprocessor directive and its relationship to the external storage classes it replaces, see *Section 5.4.5, "#pragma extern_model Directive"*.

For a description of the `_align` storage-class modifier, see *Section 4.4.3, "The _align Modifier"*.

For a description of the /EXTERN_MODEL and /[NO]SHARE_GLOBALS command-line qualifiers, see *Section 1.3.4, "CC Command Qualifiers"*.

4.3. Global Storage Classes

In addition to the storage-class specifiers described in the *VSI C Reference Manual* [https://docs.vmssoftware.com/vsi-c-language-reference-manual/], the VAX C compatibility mode of VSI C provides the `globaldef`, `globalref`, and `globalvalue` storage-class specifiers. These specifiers allow you to assign the global storage classes to identifiers. The global storage classes are specific to VSI C for OpenVMS systems and are not portable.

4.3.1. The globaldef and globalref Specifiers

Use the `globaldef` specifier to define a global variable. Use the `globalref` specifier to refer to a global variable defined elsewhere in the program.

When you use the `globaldef` specifier to define a global symbol, the symbol is placed in one of three program sections: the $DATA (VAX only) or $DATA$ (Alpha, I64) psect using `globaldef` alone, the $CODE (VAX only) or $READONLY$ (Alpha, I64) psect using `globaldef` with `readonly` or `const`, or a user-named psect. You can create a user-named psect by specifying the psect name as a string constant in braces immediately following the `globaldef` keyword, as shown in the following definition:

```
globaldef{"psect_name"}  int x = 2;
```

This definition creates a program section called `psect_name` and allocates the variable `x` in that psect. You can add any number of global variables to this psect by specifying the same psect name in other `globaldef` declarations. In addition, you can specify the `noshare` modifier to create the psect with the NOSHR attribute. Similarly, you can specify the `readonly` or `const` modifier to create the psect with the NOWRT attribute. For more information about the possible combinations of specifiers and modifiers, and the effects of the storage-class modifiers on program section attributes, see *Section 4.8, "Program Sections"*.

Variables declared with `globaldef` can be initialized; variables declared with `globalref` cannot, because these declarations refer to variables defined, and possibly initialized, elsewhere in the program. Initialization is possible only when storage is allocated for an object. This distinction is especially important when the `readonly` or `const` modifier is used; unless the global variable is initialized when the variable is defined, its permanent value is 0.

Note

In the VAX MACRO programming language, it is possible to give a global variable more than one name. However, in VSI C, only one global name can be used for a particular variable. VSI C assumes that distinct global variable names denote distinct objects; the storage associated with different names must not overlap.

Example 4.1, "Using Global Variables" shows the use of global variables.

Example 4.1. Using Global Variables

```
/*  This example shows how global variables are used      *
 *  in VSI C programs.                                     */

#include <stdlib.h>

#include <stdio.h>
extern void fn();


❶int ex_counter = 0;
❷globaldef double velocity = 3.0e10;
❸globaldef {"distance"} long miles = 100;

int main()
{
   printf("   *** FIRST COMP UNIT ***\n");
   printf("counter:\t%d\n", ex_counter);
   printf("velocity:\t%g\n", velocity);
   printf("miles:\t\t%d\n\n", miles);
   fn();
   printf("   *** FIRST COMP UNIT ***\n");
   printf("counter:\t%d\n", ex_counter);

❶   printf("velocity:\t%g\n", velocity);
   printf("miles:\t\t%d\n\n", miles);
exit (EXIT_SUCCESS);
}

/*  ---------------------------------------------------  *
 *  The following code is contained in a separate        *
 *  compilation unit.                                    *
 *  ---------------------------------------------------  */


#include <stdio.h>

static ex_counter;
❷globalref double velocity;
globalref long miles;
```

```
fn(void)
{
   ++ex_counter;
   printf("   *** SECOND COMP UNIT ***\n");
   if ( miles > 50 )
      velocity = miles * 3.1 / 200 ;
   printf("counter:\t%d\n", ex_counter);
   printf("velocity:\t%g\n", velocity);
   printf("miles:\t\t%d\n", miles);
}
```

Key to *Example 4.1, "Using Global Variables"*:

❶ In the first compilation unit, the `ex_counter` integer variable has a storage class of `extern`. In the second compilation unit, a variable named `ex_counter` is of storage class `static`. Even though they have the same identifier, the two `ex_counter` variables are different variables represented by two separate memory locations. The link-time scope of the second `ex_counter` is the module created from the second compilation unit. When control returns to the main function, the `ex_counter` external variable retains its original value.

❷ The variable `velocity` has storage class `globaldef` and is stored in the $DATA psect (VAX only) or $DATA$ psect (Alpha, I64).

❸ The `miles` variable also has storage class `globaldef` but is stored in the user-specified psect `"distance"`.

❶ When the `velocity` variable prints after the function `fn` executes, the value will have changed. Global variables have only one storage location.

❷ When you reference global variables in another module, you must declare those variables in that module. In the second module, the global variables are declared with the `globalref` keyword.

Sample output from *Example 4.1, "Using Global Variables"* is as follows:

```
$ RUN EXAMPLE.EXE
   *** FIRST COMP UNIT ***
counter:        0
velocity:       3.000000e+10
miles:          100
   *** SECOND COMP UNIT ***
counter:        1
velocity:       1.55
miles:          100
   *** FIRST COMP UNIT ***
counter:        0
velocity:       1.55
miles:          100
```

4.3.2. Comparing the Global and the External Storage Classes

The global storage-class specifiers define and declare objects that differ from external variables both in their storage allocation and in their correspondence to elements of other languages. Global variables

provide a convenient and efficient way for a VSI C function to communicate with assembly language programs, with OpenVMS system services and data structures, and with other high-level languages that support global symbol definition, such as PL/I. For more information about multilanguage programming, see *Chapter 3, "Using VSI C in the Common Language Environment"*.

VSI C imposes no limit on the number of external variables in a single program.

There are other functional differences between the external and global variables. For example:

- If you have a limited amount of storage available, you may use the `globalvalue` specifier (see *Section 4.3.3, "The globalvalue Specifier"*) since an object defined as a `globalvalue` does not occupy storage in your program; the external variables create program sections.

- You can declare a global variable, using `globaldef`, inside a function or block, and by using a `globalref` specifier, access the identifier from another compilation unit. With external variables, you must define the variable outside all functions and blocks, and then access that variable in other compilation units by using `extern` declarations.

- The global variables correspond to global symbols declared in assembly language programs, but external variables (`extern`) correspond with FORTRAN common blocks.

- A `globalref` declaration causes the linker to load the module containing the corresponding `globaldef` into the image (unless the `globalref` is not referenced, in which case VSI C optimizes it away). An `extern` declaration does not cause the linker to do so. An `extern` declaration causes an overlaying of a psect (see *Section 4.8, "Program Sections"* for details about psects).

  In programming environments other than the OpenVMS environment, C programmers may be accustomed to `extern` declarations causing the loading of a module into the program's executable image. If transportability is an issue, you can define the following symbols—at the compilation-unit level, outside of all functions—to allocate storage differently depending on the system you are using:

```
#ifdef   __DECC
#define  EXPORT  globaldef
#define  IMPORT  globalref
#else
#define  EXPORT
#define  IMPORT  extern
#endif
   .
   .
   .
IMPORT int foo;
EXPORT int foo = 53;
```

One similarity between the external and global storage classes is in the way the compiler recognizes these variables internally. External and global identifiers are not case-sensitive. No matter how the external and global identifiers appear in the source code, the compiler converts them to uppercase letters. For ease in debugging programs, express all global and external variable identifiers in uppercase letters.

Another similarity between the external and global storage classes is that you can place the external variables and the global variables (optionally) in psects with a user-defined name and, to some degree, user-defined attributes. The compiler places external variables in psects of the same name as the variable identifier, viewed by the linker in uppercase letters. The compiler places `globaldef{“name”}` variables in psects with names specified in quotation marks, delimited by braces, and located directly

after the `globaldef` specifier in a declaration. Again, the linker considers the psect name to be in uppercase letters.

The compiler places a variable declared using only the `globaldef` specifier and a data-type keyword into the $DATA (VAX only) or $DATA$ (Alpha, I64) psect. For more information about the possible combinations of specifiers and modifiers, and the effects of the storage-class modifiers on program section attributes, see *Section 4.8, "Program Sections"*.

4.3.3. The globalvalue Specifier

A global value is an integral value whose identifier is a global symbol. Global values are useful because they allow many programmers in the same environment to refer to values by identifier, without regard to the actual value associated with the identifier. The actual values can change, as dictated by general system requirements, without requiring changes in all the programs that refer to them. If you make changes to the global value, you only have to recompile the defining compilation unit (unless it is defined in an object library), not all of the compilation units in the program that refer to those definitions.

Note

You can use the `globalvalue` specifier only with identifiers of type `enum`, `int`, or with pointer variables.

An identifier declared with `globalvalue` does not require storage. Instead, the linker resolves all references to the value. If an initializer appears with `globalvalue`, the name defines a global symbol for the given initial value. If no initializer appears, the `globalvalue` construct is considered a reference to some previously defined global value.

Predefined global values serve many purposes in OpenVMS system programming, such as defining status values. It is customary in OpenVMS system programming to avoid explicit references to such values as those returned by system services, and to use instead the global names for those values.

4.4. Storage-Class Modifiers

VSI C for OpenVMS systems provides support for the storage-class modifiers `noshare`, `readonly`, and `_align` as VAX C keywords. The recognition of these three storage-class modifiers as keywords (along with the other VAX C specific keywords) is controlled by a combination of the compiler mode and the /ACCEPT command-line qualifier. The default behavior on OpenVMS systems is for the compiler to recognize these storage-class modifiers as keywords in the VAX C compatibility mode and relaxed mode (assuming that /ACCEPT=NOVAXC_KEYWORDS is not also specified.) Conversely, they are not recognized by default in all other modes unless overridden by /ACCEPT=VAXC_KEYWORDS.

VSI C also provides the `__inline`, `__forceinline` and `__align` storage-class modifiers. These are recognized as valid keywords in all compiler modes on all platforms. They are in the namespace reserved to the C implementation, so it is not necessary to allow them to be treated as user-declared identifiers. They have the same effects on all platforms, except that on VAX systems, the `__forceinline` modifier does not cause any more inlining than the `__inline` modifier does.

VSI C also provides the `inline` storage-class modifier. This modifier is supported in relaxed mode (/STANDARD=RELAXED) or if the /ACCEPT=C99_KEYWORDS or /ACCEPT=GCCINLINE qualifier is specified.

For additional information about the `__inline`, `__forceinline`, `__align`, and `inline` storage-class modifiers, see the *VSI C Reference Manual* [https://docs.vmssoftware.com/vsi-c-language-reference-manual/].

You can use a storage-class specifier and a storage-class modifier in any order; usually, the modifier is placed after the specifier in the source code. For example:

```
extern  noshare  int  x;

   /*  Or, equivalently…  */

int  noshare  extern  x;
```

The following sections describe each of the VSI C storage-class modifiers.

4.4.1. The noshare Modifier

The `noshare` storage-class modifier assigns the attribute NOSHR to the program section of the variable. Use this modifier to allow other programs, used as shareable images, to have a copy of the variable's psect without the shareable images changing the variable's value in the original psect.

When a variable is declared with the `noshare` modifier and a shared image that has been linked to your program refers to that variable, a copy is made of the variable's original psect to a new psect in the other image. The other program may alter the value of that variable within the local psect without changing the value still stored in the psect of the original program.

For example, if you need to establish a set of data that will be used by several programs to initialize local data sets, then declare the external variables using the `noshare` specifier in a VSI C program. Each program receives a copy of the original data set to manipulate, but the original data set remains for the next program to use. If you define the data as `extern` without the `noshare` modifier, a copy of the psect of that variable is not made; each program would be allowed access to the original data set, and the initial values would be lost as each program stores the values for the data in the psect. If the data is declared as `const` or `readonly`, each program is able to access the original data set, but none of the programs can then change the values.

You can use the `noshare` modifier with the `static`, `extern`, `globaldef`, and `globaldef{“name”}` storage-class specifiers. For more information about the possible combinations of specifiers and modifiers, and the effects of the storage-class modifiers on program-section attributes, see *Section 4.8, "Program Sections"*.

You can use `noshare` alone, which implies an external definition of storage class `extern`. Also, when declaring variables using the `extern` and `globaldef{“name”}` storage-class specifiers, you can use `noshare`, `const`, and `readonly`, together, in the declaration. If you declare variables using the `static` or the `globaldef` specifiers, and you use both of the modifiers in the declaration, the compiler ignores `noshare` and accepts `const` or `readonly`.

4.4.2. The readonly Modifier

The `readonly` storage-class modifier, like the `const` data-type qualifier, assigns the NOWRT attribute to the variable's program section; if used with the `static` or `globaldef` specifier, the variable is stored in the $CODE psect, which has the NOWRT attribute by default.

You can use both the `readonly` and `const` modifiers with the `static`, `extern`, `globaldef`, and `globaldef {“psect”}` storage-class specifiers.

In addition, both the `readonly` modifier and the `const` modifier can be used alone. When you specify these modifiers alone, an external definition of storage class `extern` is implied.

The `const` modifier restricts access to data in the same manner as the `readonly` modifier. However, in the declaration of a pointer, the `readonly` modifier cannot appear between the asterisk and the pointer variable to which it applies.

The following example shows the similarity between the `const` and `readonly` modifiers. In both instances, the `point` variable represents a constant pointer to a nonconstant integer.

```
readonly int * point;

int * const point;
```

Note

For new program development, VSI recommends that you use the `const` modifier, because `const` is standard-conforming and `readonly` is not.

4.4.3. The _align Modifier

The `_align` and `__align` storage-class modifiers have the same semantic meaning. The difference is that `__align` is a keyword in all compiler modes while `_align` is a keyword only in modes that recognize VAX C keywords. For new programs, using `__align` is recommended.

The `_align` and `__align` storage-class modifiers align objects of any of the VSI C data types on a specified storage boundary. Use these modifiers in a data declaration or definition.

See the *VSI C Reference Manual* [https://docs.vmssoftware.com/vsi-c-language-reference-manual/] for a detailed description of `__align` and `_align`.

4.5. Floating-Point Numbers (float, double, long double)

When declaring floating-point variables, you determine the amount of precision needed for the stored object. In VSI C, you can have single-precision, double-precision, and extended double-precision variables.

The `float` keyword declares a single-precision, floating-point variable. A `float` variable is represented internally in the VAX compatible, F_floating-point binary format.

For double-precision variables, you can choose D_floating or G_floating. On Alpha and IA-64 systems, you can also choose single- and double-precision IEEE formats (S_floating and T_floating, respectively), and extended double-precision format (X_floating).

The `double` keyword declares a double-precision, floating-point variable. VSI C provides two VAX C compatible formats for specifying `double` variables: D_floating or G_floating.

The G_floating precision of approximately 15 digits is less than that of variables represented in D_floating format. Although there are more bits allocated to the exponent in G_floating precision, fewer bits are allocated to the mantissa, which determines precision (see *Table 4.2, "Floating-Point Formats"*).

Note

When the compiler is run with the /STANDARD=VAXC qualifier, the use of the `long float` keyword, which is interchangeable with the `double` keyword, is allowed but elicits a warning that this is obsolete usage. The `long float` keyword is not allowed when the compiler is run with the /STANDARD=ANSI89 qualifier.

In VAX C, the default representation of `double` variables is D_floating. To select the G_floating representation, compile with the /G_FLOAT qualifier.

In VSI C, the /FLOAT qualifier replaces the /G_FLOAT qualifier, but /G_FLOAT is retained for compatibility.

When compiling with VSI C on OpenVMS VAX systems, if you omit both /G_FLOAT and /FLOAT, the default representation of `double` variables is D_floating (unless /MIA is specified, in which case the default is G_floating).

When compiling with VSI C on OpenVMS Alpha systems, if you omit both /G_FLOAT and /FLOAT, the default representation of `double` variables is G_floating.

When compiling with VSI C on OpenVMS IA-64 systems, the default representation of `single` and `double` variables is IEEE_floating. See the /FLOAT qualifier for more information on floating-point representation on IA-64 systems.

For OpenVMS Alpha and IA-64 systems, the /FLOAT qualifier accepts the additional option IEEE_FLOAT. If you specify /FLOAT=IEEE_FLOAT, single and `double` variables are represented in IEEE_floating format (S_floating for single float, and T_floating for `double` float).

You cannot specify both the /FLOAT and /G_FLOAT qualifiers on the command line.

Note

The VAX D_floating double-precision floating-point type is minimally supported on OpenVMS Alpha and IA-64 systems. When compiling with this type, all data transfer is done with the data in D_floating format, but for each arithmetic operation the data is converted first to G_floating and then back to D_floating format when the operation is complete. Therefore, it is possible to lose three binary digits of precision in arithmetic operations. This floating-point type is provided for compatibility with VAX systems.

Modules compiled with the D_floating representation should not be linked with modules compiled with the G_floating representation. Since there are no functions in the VSI C Run-Time Library (C RTL) that perform floating-point format conversions on files, use the OpenVMS RTL functions MTHCVT_D_G, MTHCVT_G_D, MTH$CVT_DA_GA, and MTH$CVT_GA_DA if you do not wish to recompile the program. For more information about using the OpenVMS RTL, see the *VMS Run-Time Library Routines Volume*.

On VAX systems, VSI C maps the standard C defined `long double` type to the G_floating or D_floating format.

On OpenVMS Alpha and IA-64 systems, `long double` variables are represented by default in the software-emulated X_floating format. If you specify /L_DOUBLE_SIZE=64, `long double` variables

are represented as G_floating, D_floating, or T_floating, depending on the value of the /FLOAT or /G_FLOAT qualifier.

Note

Modules must be linked to the appropriate run-time library. For more information about linking against the VSI C RTL shareable image and object libraries, see the *VSI C Run-Time Library Reference Manual for OpenVMS Systems* [https://docs.vmssoftware.com/vsi-c-run-time-library-reference-manual-for-openvms-systems/].

Table 4.2, "Floating-Point Formats" shows the supported floating-point formats, and their approximate sizes and range of values.

Table 4.2. Floating-Point Formats

Data type	Floating-Point Format	Length of Variable	Range of Values	Precision (decimal digits)
`float`	F_floating	32-bit	$2.9 * 10^{-39}$ to $1.7 * 10^{38}$	6
`double`	D_floating	64-bit	$2.9 * 10^{-39}$ to $1.7 * 10^{38}$	16
`double`	G_floating	64-bit	$5.6 * 10^{-309}$ to $9.0 * 10^{307}$	15
`float`	S_floating (Alpha, I64)	32-bit	$1.2 * 10^{-38}$ to $3.4 * 10^{38}$	6
`double`	T_floating (Alpha, I64)	64-bit	$2.2 * 10^{-308}$ to $1.8 * 10^{308}$	15
`long double`	X_floating (Alpha, I64)	128-bit	$3.4 * 10^{-4932}$ to $1.2 * 10^{4932}$	33

4.6. Pointer Conversions

When running the compiler in VAX C mode, relaxed pointer and pointer/integer compatibility is allowed. That is, all pointer and integer types are compatible, and pointer types are compatible with each other regardless of the type of the object they point to. Therefore, in VAX C mode, a pointer to `float` is compatible with a pointer to `int`. This is not true in ANSI C mode.

Although pointer conversions do not involve a representation change when compiling in VAX C mode, because of alignment restrictions on some machines, access through an unaligned pointer can result in much slower access time, a machine exception, or unpredictable results.

4.7. Structure Alignment

The alignment and size of a structure is affected by the alignment requirements and sizes of the structure components for each VSI C platform. A structure can begin on any byte boundary and occupy any integral number of bytes. However, individual architectures or operating systems can specify particular alignment and padding requirements.

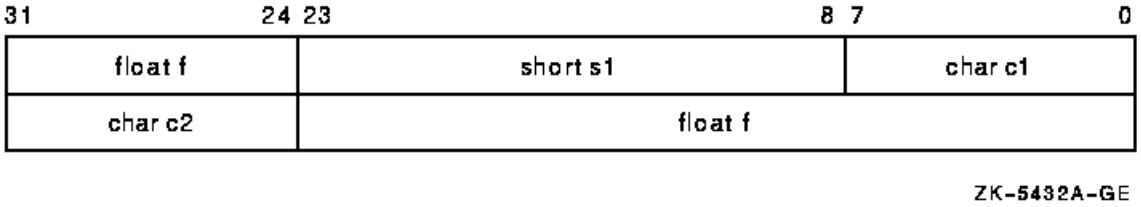
VSI C on VAX processors does not require that structures or structure members be aligned on any particular boundaries.

The components of a structure are laid out in memory in the order they are declared. The first component has the same address as the entire structure. On VAX processors, each additional component follows its predecessor in the immediately following byte.

For example, the following type is aligned as shown in *Figure 4.1, "VAX Structure Alignment"*:

```
struct {char c1;
        short s1;
        float f;
        char c2;
        }
```

Figure 4.1. VAX Structure Alignment



The alignment of the entire structure can occur on any byte boundary, and no padding is introduced. The `float` variable `f` may span longwords, and the `short` variable `s1` may span words.

The following pragma can be used to force specific alignments:

```
#pragma member_alignment
```

Structure alignment for VSI C for OpenVMS systems on VAX processors is achieved by the default, `#pragma nomember_alignment`, which causes data structure members to be byte-aligned (with the exception of bit-field members).

Structure alignment for VSI C for OpenVMS systems on Alpha and Itanium processors is achieved by the default, `#pragma member_alignment`, which causes data structure members to be naturally aligned. This means that data structure members are aligned on the next boundary appropriate to the type of the member, rather than on the next byte.

For more information on the `#pragma member_alignment` preprocessor directive, see *Section 5.4.13, "#pragma [no]member_alignment Directive"*.

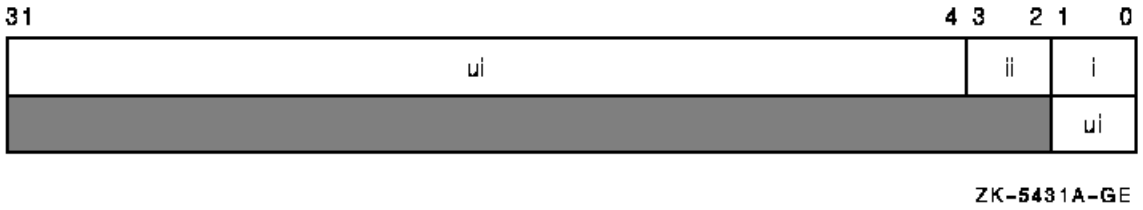
4.7.1. Bit-Field Alignment

Bit fields can have any integral type. However, the compiler issues a warning if /STANDARD=ANSI89 is specified, and the type is something other than `int`, `unsigned int`, or `signed int`. Bit fields are allocated within the unit from low order to high order. If a bit field immediately follows another bit field, the bits are packed into adjacent space, even if this overflows into another byte. However, if an unnamed bit field is specified to have length 0, filler is added so the bit field immediately following starts on the next byte boundary.

For example, the following type is aligned as shown in *Figure 4.2, "OpenVMS Bit-Field Alignment"*:

```
struct {int i:2;
        int ii:2;
        unsigned int ui: 30;
        }
```

Figure 4.2. OpenVMS Bit-Field Alignment



Bit field `ii` is positioned immediately following bit field `i`. Because there are only 28 bit positions remaining and `ui` requires 30 bits, the first 28 bits of `ui` are put into the first longword, and the remaining two bits overflow into the next longword.

4.7.2. Bit-Field Initialization

The VSI C compiler initializes bit fields in `structs` differently than VAX C does. The following program compiles without error using both compilers but the results are different. VSI C skips over unnamed bits but VAX C does not.

```
#include <stdio.h>

int t()
{
   static struct bar {unsigned :1;
               unsigned one : 1;
               unsigned two : 1;
              };
   struct bar foo = {1,0};
   printf("%d %d\n",foo.one,foo.two);
   return 1;
}
```

When compiled with VSI C, this example produces the following output:

```
1 0
```

When compiled with VAX C, this example produces the following output:

```
0 0
```

4.7.3. Variant Structures and Unions

Variant structures and unions are VSI C extensions available in VAX C compatibility mode only, and they are not portable.

Variant structure and union declarations allow you to refer to members of nested aggregates without having to refer to intermediate structure or union identifiers. When a variant structure or union declaration is nested within another structure or union declaration, the enclosed variant aggregate ceases to exist as a separate aggregate, and VSI C propagates its members to the enclosing aggregate.

Variant structures and unions are declared using the `variant_struct` and `variant_union` keywords. The format of these declarations is the same as that for regular structures or unions, with the following exceptions:

- Variant aggregates must be nested within other valid structure or union declarations.

- A tag cannot be used in a variant aggregate declaration.
- At least one declarator must be declared in the variant aggregate declaration, and it must not be declared as a pointer or an array.

Initialization of a variant structure or union is the same as that for a normal structure or union.

As with regular structures and unions, in VAX C compatibility mode, variant structures and unions in an assignment operation need only have the same size in bits, rather than requiring the same members and member types.

To show the use of variant aggregates, consider the following code example that does not use variant aggregates:

```
/*  The numbers to the right of the code represent the byte offset  *
 *  from the enclosing structure or union declaration.              */
struct  TAG_1
{
   int   a;           /* 0-byte  enclosing_struct offset */
   char  *b;          /* 4-byte  enclosing_struct offset */
   union  TAG_2       /* 8-byte  enclosing_struct offset */
   {
      int  c;         /* 0-byte  nested_union offset */
      struct  TAG_3   /* 0-byte  nested_union offset */
      {
         int  d;      /* 0-byte  nested_struct offset */
         int  e;      /* 4-byte  nested_struct offset */
      }  nested_struct;
   } nested_union;
} enclosing_struct;
```

If you want to access nested member `d`, then you need to specify all the intermediate aggregate identifiers:

```
enclosing_struct.nested_union.nested_struct.d
```

If you try to access member `d` without specifying the intermediate identifiers, then you would access the incorrect offset from the incorrect structure. Consider the following example:

```
enclosing_struct.d
```

The compiler uses the address of the original structure (`enclosing_struct`), and adds to it the assigned offset value for member `d` (0 bytes), even though the offset value for `d` was calculated according to the nested structure (`nested_struct`). Consequently, the compiler accesses member `a` (0-byte offset from `enclosing_struct`) instead of member `d`.

The following code example shows the same code using variant aggregates:

```
/*  The numbers to the right of the code present the byte offset  *
 *  from enclosing_struct.                                        */
struct  TAG_1
{
   int   a;           /* 0-byte  enclosing_struct offset */
   char  *b;          /* 4-byte  enclosing_struct offset */
   variant_union
   {
      int  c;         /* 8-byte  enclosing_struct offset */
      variant_struct
      {
```

```
         int  d;      /* 8-byte   enclosing_struct offset */
         int  e;      /* 12-byte  enclosing_struct offset */
      }  nested_struct;
   } nested_union;
} enclosing_struct;
```

The members of the `nested_union` and `nested_struct` variant aggregates are propagated to the immediately enclosing aggregate (`enclosing_struct`). The variant aggregates cease to exist as individual aggregates.

Since the `nested_union` and `nested_struct` variant aggregates do not exist as individual aggregates, you cannot use tags in their declarations, and you cannot use their identifiers (`nested_union`, `nested_struct`) in any reference to their members. However, you are free to use the identifiers in other declarations and definitions within your program.

To access member `d`, use the following notation:

```
enclosing_struct.d
```

Using the following notation causes unpredictable results:

```
enclosing_struct.nested_union.nested_struct.d
```

If you use normal structure or union declarations within a variant aggregate declaration, the compiler propagates the structure or union to the enclosing aggregate, but the members remain a part of the nested aggregate. For example, if the nested structure in the last example was of type `struct`, the following offsets would be in effect:

```
struct  TAG_1
{
   int   a;           /* 0-byte  enclosing_struct offset */
   char  *b;          /* 4-byte  enclosing_struct offset */
   variant_union
   {
      int  c;         /* 8-byte  enclosing_struct offset */
      struct  TAG_2   /* 8-byte  enclosing-struct offset */
      {
         int  d;      /* 0-byte   nested_struct offset */
         int  e;      /* 4-byte   nested_struct offset */
      }  nested_struct;
   } nested_union;
} enclosing_struct;
```

In this case, to access member `d`, use the following notation:

```
enclosing_struct.nested_union.nested_struct.d
```

4.8. Program Sections

The following sections describe program-section attributes and program sections created by VSI C for OpenVMS systems.

4.8.1. Attributes of Program Sections

As the VSI C compiler creates an object module, it groups data into contiguous program sections, or *psects*. The grouping depends on the attributes of the data and on whether the psects contain executable code or read/write variables.

The compiler also writes into each object module information about the program sections contained in it. The linker uses this information when it binds object modules into an executable image. As the linker allocates virtual memory for the image, it groups together program sections that have similar attributes.

Table 4.3, "Program-Section Attributes" lists the attributes that can be applied to program sections.

Table 4.3. Program-Section Attributes

Attribute	Meaning
PIC or NOPIC	The program section or the data these attributes refers to does not depend on any specific virtual memory location (PIC), or else the program section depends on one or more virtual memory locations (NOPIC).[1]
CON or OVR	The program section is concatenated with other program sections with the same name (CON) or overlaid on the same memory locations (OVR).
REL or ABS	The data in the program section can be relocated within virtual memory (REL) or is not considered in the allocation of virtual memory (ABS).
GBL or LCL	The program section is part of one cluster, is referenced by the same program section name in different clusters (GBL), or is local to each cluster in which its name appears (LCL).
EXE or NOEXE	The program section contains executable code (EXE) or does not contain executable code (NOEXE).
WRT or NOWRT	The program section contains data that can be modified (WRT) or data that cannot be modified (NOWRT).
RD or NORD	These attributes are reserved for future use.
SHR or NOSHR	The program section can be shared in memory (SHR) or cannot be shared in memory (NOSHR).
USR or LIB	These attributes are reserved for future use.
VEC or NOVEC	The program section contains privileged change mode vectors (VEC) or does not contain those vectors (NOVEC).
COM or NOCOM	The program section is a conditionally defined psect associated with a conditionally defined symbol. This is the type of psect created when you declare an uninitialized definition with `extern_model relaxed_refdef`.

[1]VSI C programs can be bound into PIC or NOPIC shareable images. NOPIC occurs if declarations such as the following are used: *char *x = &y;*. This statement relies on the address of variable *y* to determine the value of the pointer *x*.

4.8.2. Program Sections Created by VSI C

If necessary, VSI C creates the following program sections:

- $CODE (VAX only)—Contains all executable code and constant data (including variables defined with the `readonly` modifier or `const` type qualifier).
- $CODE$ (Alpha, I64)—Contains all executable code.

- $READONLY$ (Alpha, I64)—Contains all constant data defined with the `readonly` modifier or `const` type qualifier.
- $DATA (VAX only) or $DATA$ (Alpha, I64)—Contains all static variables, as well as global variables defined without the `readonly` modifier or `const` type qualifier. $DATA also contains character-string constants when /ASSUME=WRITABLE_STRING_LITERALS is specified.
- $LITERAL$ (Alpha, I64)—Contains character-string constants when /ASSUME=NOWRITABLE_STRING_LITERALS is specified.
- VSI C also creates additional program sections for variables declared with the `globaldef` keyword if the optional psect name in braces is specified, or for variables declared with the `extern` storage class, depending on the external model.

All program sections created by VSI C have the PIC, REL, RD, USR, and NOVEC attributes. On VAX systems, the $CODE psect is aligned on a byte boundary; all other psects generated by VSI C are aligned on longword boundaries. On OpenVMS Alpha and IA-64 systems, all psects generated by VSI C are aligned on octaword boundaries. Note that use of the `_align` storage-class modifier can cause a psect to be aligned on greater than a longword boundary on OpenVMS VAX systems. The $CHAR_STRING_CONSTANTS psect has the same attributes as the $DATA (VAX only) and $DATA$ (Alpha, I64) psects.

Tables *Table 4.4, "External Models and Definitions"*, *Table 4.5, "Combinations of Storage-Class Specifiers and Modifiers (Alpha, I64)"*, *Table 4.6, "Combinations of Storage-Class Specifiers and Modifiers (VAX only)"*, and *Table 4.7, "Combination Attributes"* summarize the differences in psects created by different declarations:

- *Table 4.4, "External Models and Definitions"*, *Table 4.5, "Combinations of Storage-Class Specifiers and Modifiers (Alpha, I64)"* (Alpha, I64), and *Table 4.6, "Combinations of Storage-Class Specifiers and Modifiers (VAX only)"* (VAX only) show different cases of variable definitions and assign to them a storage-class code number:
  - *Table 4.4, "External Models and Definitions"* shows the effect of each `#pragma extern_model` preprocessor directive on the storage-class code number for external variable definitions that have an `extern` storage class.
  - *Table 4.5, "Combinations of Storage-Class Specifiers and Modifiers (Alpha, I64)"* shows the storage-class code number for variable definitions that do not have the `extern` storage class on OpenVMS Alpha and IA-64 systems.
  - *Table 4.6, "Combinations of Storage-Class Specifiers and Modifiers (VAX only)"* shows the storage-class code number for variable definitions that do not have the `extern` storage class on VAX systems.
- *Table 4.7, "Combination Attributes"* shows the psect name and attributes associated with each storage-class code number from Tables *Table 4.4, "External Models and Definitions"*, *Table 4.5, "Combinations of Storage-Class Specifiers and Modifiers (Alpha, I64)"*, and *Table 4.6, "Combinations of Storage-Class Specifiers and Modifiers (VAX only)"*.

Table 4.4. External Models and Definitions

Storage-Class Code	External Object Definition	Interpretation
External Model: `#pragma extern_model common_block noshr`		

Storage-Class Code	External Object Definition	Interpretation
1	`int name;`	/* uninitialized definition */
1	`int name = 1;`	/* initialized definition */
1	`extern int name;`	/* treated as an uninitialized definition */
2	`const int name;`	/* uninitialized definition */
2	`const int name = 1;`	/* initialized definition */
2	`extern const int name;`	/* treated as an uninitialized definition */
External Model: `#pragma extern_model common_block shr`		
3	`int name;`	/* uninitialized definition */
3	`int name = 1;`	/* initialized definition */
3	`extern int name;`	/* treated as an uninitialized definition */
4	`const int name;`	/* uninitialized definition */
4	`const int name = 1;`	/* initialized definition */
4	`extern const int name;`	/* treated as an uninitialized definition */
External Model: `#pragma extern_model relaxed_refdef noshr`		
5	`int name;`	/* uninitialized definition */
1	`int name = 1;`	/* initialized definition */
6	`const int name;`	/* uninitialized definition */
2	`const int name = 1;`	/* initialized definition */
External Model: `#pragma extern_model relaxed_refdef shr`		
7	`int name;`	/* uninitialized definition */
3	`int name = 1;`	/* initialized definition */
8	`const int name;`	/* uninitialized definition */
4	`const int name = 1;`	/* initialized definition */
External Model: `#pragma extern_model strict_refdef`		
9 (Alpha, I64)	`int symbol;`	/* uninitialized definition */
10 (VAX only)	`int symbol;`	/* uninitialized definition */
10	`int symbol = 1;`	/* initialized definition */
11	`const int symbol;`	/* uninitialized definition */
11	`const int symbol = 1;`	/* initialized definition */
External Model: `#pragma extern_model strict_refdef "name" noshr`		
12	`int symbol;`	/* uninitialized definition */
12	`int symbol = 1;`	/* initialized definition */
13	`const int symbol;`	/* uninitialized definition */

Storage-Class Code	External Object Definition	Interpretation
13	`const int symbol = 1;`	/* initialized definition */
External Model: `#pragma extern_model strict_refdef "name" shr`		
14	`int symbol;`	/* uninitialized definition */
14	`int symbol = 1;`	/* initialized definition */
15	`const int symbol;`	/* uninitialized definition */
15	`const int symbol = 1;`	/* initialized definition */

Table 4.5. Combinations of Storage-Class Specifiers and Modifiers *(Alpha, I64)*

Storage-Class Code	Storage-Class Keyword Combination	/SHARE or /NOSHARE	Initialized or Not
9	`static`	Either	No
10	`static`	Either	Yes
11	`static const`[1]	Either	Either
9	`globaldef`	Either	No
10	`globaldef`	Either	Yes
11	`globaldef const`[1]	Either	Either
14	`globaldef{"name"}`	/SHARE	Either
12	`globaldef{"name"}`	/NOSHARE	Either
15	`globaldef{"name"} const`[1]	/SHARE	Either
13	`globaldef{"name"} const`[1]	/NOSHARE	Either

[1]Using in place of produces the same results.

Table 4.6. Combinations of Storage-Class Specifiers and Modifiers *(VAX only)*

Storage-Class Code	Storage-Class Keyword Combination	/SHARE or /NOSHARE
10	`static`	Either
11	`static const`[1]	Either
10	`globaldef`	Either
11	`globaldef const`[1]	Either
14	`globaldef{"name"}`	/SHARE
12	`globaldef{"name"}`	/NOSHARE
15	`globaldef{"name"} const`[1]	/SHARE

Storage-Class Code	Storage-Class Keyword Combination	/SHARE or /NOSHARE
13	`globaldef`{"name"} `const`[1]	/NOSHARE

[1]Using in place of produces the same results.

Table 4.7, "Combination Attributes" shows the psect name and psect attributes for the storage-class code numbers from *Table 4.4, "External Models and Definitions"*, *Table 4.5, "Combinations of Storage-Class Specifiers and Modifiers (Alpha, I64)"*, and *Table 4.6, "Combinations of Storage-Class Specifiers and Modifiers (VAX only)"*. Where `name` is used for the psect name in *Table 4.7, "Combination Attributes"*, the name of the psect is the same as `name` in the declarations or pragmas in *Table 4.4, "External Models and Definitions"*, or the quoted brace-enclosed names in Tables *Table 4.5, "Combinations of Storage-Class Specifiers and Modifiers (Alpha, I64)"* and *Table 4.6, "Combinations of Storage-Class Specifiers and Modifiers (VAX only)"*.

Table 4.7. Combination Attributes

Storage-Class Code	Program Section Name	Program Attributes
1	`name`	OVR, GBL, NOSHR, NOEXE, WRT, NOCOM
2	`name`	OVR, GBL, NOSHR, NOEXE, NOWRT, NOCOM
3	`name`	OVR, GBL, SHR, NOEXE, WRT, NOCOM
4	`name`	OVR, GBL, SHR, NOEXE, NOWRT, NOCOM
5	`name`	OVR, GBL, NOSHR, NOEXE, WRT, COM
6	`name`	OVR, GBL, NOSHR, NOEXE, NOWRT, COM
7	`name`	OVR, GBL, SHR, NOEXE, WRT, COM
8	`name`	OVR, GBL, SHR, NOEXE, NOWRT, COM
9	BSS	CON, LCL, NOSHR, NOEXE, WRT, NOCOM
10	$DATA (VAX only)	CON, LCL, NOSHR, NOEXE, WRT, NOCOM
10	$DATA$ (Alpha, I64)	CON, LCL, NOSHR, NOEXE, WRT, NOCOM
11	$CODE (VAX only)	CON, LCL, SHR, EXE, NOWRT, NOCOM
11	$READONLY$ (Alpha, I64)	CON, LCL, SHR, NOEXE, NOWRT, NOCOM
12	"name"	CON, GBL, NOSHR, NOEXE, WRT, NOCOM
13	"name"	CON, GBL, NOSHR, NOEXE, NOWRT, NOCOM
14	"name"	CON, GBL, SHR, NOEXE, WRT, NOCOM
15	"name"	CON, GBL, SHR, NOEXE, NOWRT, NOCOM

The combined use of the `readonly` and `noshare` modifiers is ignored by the compiler in the following declarations:

```
readonly  noshare  static  int  x;
readonly  noshare  globaldef  int  x;
```

When it encounters a situation as shown in the previous example, the compiler ignores the `noshare` modifier and accepts `readonly`. The order of the storage-class specifier, the storage-class modifier, and the data-type keyword within a declaration is not significant.

The VSI C compiler does static (global) initialization of pointers by using the .ADDRESS directive. By using this mechanism, the compiler efficiently generates position-independent code. The linker makes image sections that contain such initialization nonshareable.

Chapter 5. Preprocessor Directives

The VSI C preprocessor provides the ability to perform macro substitution, conditional compilation, and inclusion of named files. Preprocessor directives, lines beginning with # and possibly preceded by white space, are used to communicate with the preprocessor. The *VSI C Reference Manual* [https://docs.vmssoftware.com/vsi-c-language-reference-manual/] describes the standard-conforming preprocessor directives available with the VSI C compiler. This chapter describes the preprocessor directives that are either specific to VSI C on OpenVMS systems, or that are used in an implementation-specific way:

- The `#dictionary` directive, used for CDD/Repository extraction (*Section 5.4.3, "#pragma dictionary Directive"*, *Section 5.1, "CDD/Repository Extraction (#dictionary)"*)
- The `#include` directive, used for file inclusion (*Section 5.2, "File Inclusion (#include)"*)
- The `#module` directive, for specifying an alternative name and identification for the object module (*Section 5.3, "Changing the Default Object Module Name and Identification (#module)"*, *Section 5.4.15, "#pragma module Directive"*)
- The `#pragma` directive and pragmas specific to OpenVMS systems (*Section 5.4, "Implementation-Specific Preprocessor Directive (#pragma)"*)

If you plan to port programs to and from other C implementations, take care in choosing which preprocessor directives to use within your programs. See the *VSI C Reference Manual* [https://docs.vmssoftware.com/vsi-c-language-reference-manual/] for more information about using preprocessor directives for conditional compilation. For a complete discussion of portability concerns, see the *VSI C Run-Time Library Reference Manual for OpenVMS Systems* [https://docs.vmssoftware.com/vsi-c-run-time-library-reference-manual-for-openvms-systems/].

Preprocessor directives are independent of the usual scope rules; they remain in effect from their occurrence until the end of the compilation unit. For more information about the compilation unit, see *Chapter 1, "Developing VSI C Programs"*.

5.1. CDD/Repository Extraction (#dictionary)

The `#dictionary` directive is retained for compatibility with VAX C, and is supported only when running VSI C in VAX C mode (/STANDARD=VAXC). See *Section 5.4.3, "#pragma dictionary Directive"* for information on using the standard C equivalent `#pragma dictionary` directive.

5.2. File Inclusion (#include)

The `#include` directive inserts external text into the source stream delivered to the compiler. This directive is often used to include global definitions for use with VSI C functions and macros in the program text.

The `#include` directive is supported on all VSI C implementations, but the syntax and semantics vary. For example, the directory search algorithm for locating included files on OpenVMS systems differs from that on UNIX systems, primarily because of differences in the native file systems and conventions on the two platforms. Nevertheless, by choosing the lowest common denominator of plain text files in directories to contain header files, you can define command-line options for both platforms

to cause searching to be done in the same way. VSI C for OpenVMS systems also provides a form of the `#include` directive specifically for including text modules from OpenVMS text library files. The following sections describe the `#include` directive as implemented on OpenVMS systems.

The `#include` directives may be nested to a depth determined by the FILLM process quota and by virtual memory restrictions. The VSI C compiler imposes no inherent limitation on the nesting level of inclusion.

OpenVMS and most UNIX style file specifications can be included in VSI C source programs.

The following sections describe the different forms of the `#include` directive.

5.2.1. Inclusion Using Angle Brackets

The first form of the `#include` preprocessor directive uses angle brackets `(<>)` to delimit the file specification:

```
#include <file-spec>
```

The *file-spec* is a valid file specification or a logical name. A file specification may be up to 255 characters long.

If the *file-spec* contains "/" or "!" characters, it is assumed to be a UNIX style name, and the compiler attempts to combine it with other UNIX style names from the /INCLUDE_DIRECTORY command-line qualifier and translate the result to an OpenVMS file specification using RTL functions. Otherwise, the *file-spec* is treated as an OpenVMS file specification with defaults supplied from command-line qualifiers and logical names in a prescribed search order.

When specifying the names of files to be included in your source program, avoid directory specifications of the following form:

```
DBA0:[.dir-name…]
```

Depending on device logical names is not good practice. Instead, try to use only simple file names complete with the .h file type, and use the /INCLUDE_DIRECTORY qualifier to specify the directories to search.

For the angle-bracket form of inclusion, the compiler searches directories in the following order for the file to be included:

1. Any directories specified with the /INCLUDE_DIRECTORY qualifier.
2. The directory or search list of directories specified in the logical name DECC$SYSTEM_INCLUDE, if DECC$SYSTEM_INCLUDE is defined.
3. If DECC$SYSTEM_INCLUDE is not defined, then the directory or search list of directories specified by DECC$LIBRARY_INCLUDE.
4. If neither DECC$SYSTEM_INCLUDE nor DECC$LIBRARY_INCLUDE are defined as logical names, the compiler searches the following directories for plain text-file copies of compiler header files:

   SYS$COMMON:[DECC$LIB.INCLUDE.DECC$RTLDEF]
   SYS$COMMON:[DECC$LIB.INCLUDE.SYS$STARLET_C]

   Normally, the compiler installation does not put any files in these directories, but the compiler will search them if they exist.

5. If the file is still not found, all directories and the file extension are stripped off and the steps for including a module from a text library are followed.

6. If the file is still not found, SYS$LIBRARY is searched.

You can define DECC$SYSTEM_INCLUDE to be a valid directory specification or a search list of valid directory specifications. Before each compilation of your program, you can redefine DECC$SYSTEM_INCLUDE to be any valid directory or list of directories you choose.

Avoid defining DECC$SYSTEM_INCLUDE to be a rooted directory or subdirectory of the following form:

```
DBA0:[dir-name.]
```

When defining DECC$SYSTEM_INCLUDE, use complete directory specifications.

If DECC$SYSTEM_INCLUDE translates to a directory or a search list of directories, and if the compiler cannot locate the specified file, the compiler generates an error message. If DECC$SYSTEM_INCLUDE is undefined, the compiler then searches the DECC$LIBRARY_INCLUDE or SYS$LIBRARY directory for the specified file; if the file cannot be found, the compiler generates an error message. For more information about search lists, see the DCL command DEFINE in the *VSI OpenVMS DCL Dictionary*.

Note

The purpose of DECC$LIBRARY_INCLUDE is to identify an alternative location for *all* header files normally provided by the compiler installation. Therefore, if this logical is defined, the compiler does not search the SYS$COMMON directories, the SYS$LIBRARY text libraries, or header files it would normally search.

The purpose of DECC$SYSTEM_INCLUDE is to define the order for searching directories of plain-text files for the angle-bracketed form of `#include`. Defining this logical does not suppress the search of the SYS$LIBRARY text libraries where the compiler-supplied header files normally reside.

When porting programs to the OpenVMS environment, your programs may contain `#include` directives of the following form:

```
#include <sys/file.h>
```

The VSI C compiler translates this line, common in programs that run on UNIX systems, to the following UNIX style file specification:

```
/sys/file.h
```

The compiler then translates the UNIX style file specification to the OpenVMS file specification as follows:

```
SYS:FILE.H
```

If you port programs containing such directives, define the SYS logical to be the proper name of the OpenVMS directory containing the files to be included.

Another way to use UNIX style directories is to specify them on the /INCLUDE_DIRECTORY command-line qualifier. They must contain a "/" character and must, therefore, be in quotation marks.

5.2.2. Inclusion Using Quotation Marks

The second form of the `#include` preprocessor directive uses quotation marks to delimit the file specification:

`#include "`*`file-spec`*`"`

The *file-spec* is a valid OpenVMS or UNIX style file specification.

For this form of file inclusion, the compiler searches directories in the following order for the file to be included:

1. One of the following directories:
   - If /NESTED_INCLUDE_DIRECTORY=INCLUDE_FILE (the default) is specified, the directory where the immediately containing include file is located (that is, the directory containing the file in which the `#include` directive occurred).
   - If /NESTED_INCLUDE_DIRECTORY=PRIMARY_FILE is specified, the directory containing the top-level source file (that is, the directory containing the .C file being compiled, which is not necessarily the current default directory). This is most similar to the behavior of the VAX C compiler.
   - If /NESTED_INCLUDE_DIRECTORY=NONE is specified, then skip this step and begin at step 2.
2. Any directories specified with the /INCLUDE_DIRECTORY qualifier.
3. The directory or search list of directories specified in the logical name DECC$USER_INCLUDE, if DECC$USER_INCLUDE is defined.
4. If the file is still not found, the steps for angle-bracketed files are followed.

Note that when /NESTED_INCLUDE_DIRECTORY=PRIMARY_FILE is specified, the directory containing the top-level source file is not necessarily the current RMS default device and directory.

For example, given the current directory, DBA0:[CURRENT], and the following CC command line, the compiler searches DBA0:[OTHERDIR] for any included files delimited by quotation marks, even though the current RMS default is the directory, DBA0:[CURRENT]:

```
$ CC  DBA0:[OTHERDIR]EXAMPLE.C
```

If the compiler cannot locate the specified file, it searches any directories specified by the /INCLUDE_DIRECTORY qualifier.

If the compiler still cannot locate the specified file, it translates the logical name DECC$USER_INCLUDE. If DECC$USER_INCLUDE translates to a valid directory specification or a search list of directories, the compiler searches that directory or directories for the specified file. Before each compilation of your program, you can redefine DECC$USER_INCLUDE to be any valid directory or list of directories you choose.

As with DECC$SYSTEM_INCLUDE, do not define DECC$USER_INCLUDE to be a rooted directory or subdirectory. Use complete directory specifications when defining DECC$USER_INCLUDE.

If you defined DECC$USER_INCLUDE, and the compiler cannot locate the specified file in that directory or search list of directories, the *file-spec* is treated as if it were enclosed in angle brackets instead of quotation marks.

5.2.3. Inclusion of Text Modules

The third form of the `#include` preprocessor directive is used for including module names:

```
#include module-name
```

The *module-name* is the name of a module in a text library.

This method of inclusion is not portable unless *module-name* is a macro that expands to either the angle-bracket or quoted form. This module-name syntax is provided for compatibility with VAX C and other OpenVMS compilers only, and should generally be avoided.

VSI C text libraries on OpenVMS systems are specified and searched in the following manner:

1. A text library can be created with the LIBRARY command and specified with the /LIBRARY qualifier on the CC command line.

2. If you compile more than one compilation unit using a single CC command, you must specify the library within each of the compilation units, if needed. For example:

   ```
   $ CC sourcea+mylib/LIBRARY, sourceb+mylib/LIBRARY
   ```

3. If you specify more than one library to the VSI C compiler, and if the `#include` directives are not nested (see the note in *Section 5.2.2, "Inclusion Using Quotation Marks"*), then the libraries are searched in the specified order each time an `#include` directive is encountered. Consider the following example:

   ```
   $ CC sourcea+mylib/LIBRARY+yourlib/LIBRARY
   ```

   In this example, the compiler searches for modules referenced in `#include` directives first in MYLIB.TLB and then in YOURLIB.TLB.

4. If no library is specified on the CC command line, or if the specified module cannot be found in any of the specified libraries, the following actions are taken:

   - If you defined an equivalence name for DECC$TEXT_LIBRARY that names a text library, that library is searched.

   - The compiler searches for any remaining unresolved module names in the following location, which contains the VSI C RTL header files:

     SYS$LIBRARY:DECC$RTLDEF.TLB

     For OpenVMS Version 7.1 and higher, the compiler then searches the following location, which contains the STARLET header files:

     SYS$LIBRARY:SYS$STARLET_C.TLB

5.2.4. Macro Substitution in #include Directives

VSI C allows macro substitution within the `#include` preprocessor directive.

For example, if you want to include a file name, you can use the following two directives:

```
#define  macro1  "file.ext"
```

```
#include macro1
```

If you use defined macros in `#include` directives, the macros must evaluate to one of the three following acceptable `#include` file specifications or the use generates an error message:

```
<file-spec>
"file-spec"
module-name
```

5.3. Changing the Default Object Module Name and Identification (#module)

The `#module` directive is retained for compatibility with VAX C and is supported only when running VSI C in VAX C mode (/STANDARD=VAXC). See *Section 5.4.15, "#pragma module Directive"* for information on using the standard C equivalent `#pragma module` directive.

5.4. Implementation-Specific Preprocessor Directive (#pragma)

The `#pragma` directive is a standard method for implementing features that vary from one compiler to the next. This section describes the implementation-specific pragmas that are available on the VSI C compiler for OpenVMS systems. Pragmas supported by all implementations of VSI C are described in the *VSI C Reference Manual* [https://docs.vmssoftware.com/vsi-c-language-reference-manual/].

Some `#pragma` directives are subject to macro expansion in the preprocessor before being translated. A macro reference can occur anywhere after the keyword `pragma`. The following example demonstrates this feature using the `#pragma inline` directive:

```
#define opt inline
#define f func
#pragma opt(f)
```

The `#pragma` directive becomes `#pragma inline (func)` after both macros are expanded.

The following pragmas are subject to macro expansion:

```
builtins            inline                linkage       standard
dictionary          noinline              module        nostandard
extern_model        member_alignment      message       use_linkage
extern_prefix       nomember_alignment
```

Note

An `_nm` suffix can be appended to any of the above-listed macros to prevent macro expansion. For example, to prevent macro expansion on `#pragma inline`, specify it as `#pragma inline_nm`.

Also, to provide macro-expansion support to those pragmas not listed above, all pragmas (including those that are already specified as undergoing macro expansion) have an alternative *pragma-name*`_m` version, which makes the pragma subject to macro expansion. For example, `#pragma assert` is not subject to macro expansion, but `#pragma assert_m` is. Another example: `#pragma module` and `#pragma module_m` are equivalent and both subject to macro expansion.

The following sections describe the `#pragma` directives.

5.4.1. #pragma assert Directive

The `#pragma assert` directive lets you specify assertions that the compiler can make about a program to generate more efficient code. The pragma can also be used to verify that certain compile-time conditions are met; this is useful in detecting conditions that could cause run-time faults.

The `#pragma assert` directive is never needed to make a program execute correctly, however if a `#pragma assert` is specified, the assertions must be valid or the program might behave incorrectly.

The `#pragma assert` directive has the following formats:

```
#pragma assert func_attrs(identifier-list)function-assertions
#pragma assert global_status_variable(variable-list)
#pragma assert non_zero(constant-expression) string-literal
```

5.4.1.1. #pragma assert func_attrs

Use this form of the pragma to make assertions about a function's attributes.

The *identifier-list* is a list of function identifiers about which the compiler can make assumptions. If more than one identifier is specified, separate them by commas.

The *function-assertions* specify the assertions to the compiler about the functions. Specify one or more of the following, separating multiple assertions with white space:

```
noreturn
nocalls_back
nostate
noeffects
file_scope_vars(option)
format (style, format-index, first-to-check-index)
```

`noreturn` asserts to the compiler that any call to the routine will never return.

`nocalls_back` asserts to the compiler that no routine in the source module will be called before control is returned from this function.

`nostate` asserts to the compiler that the value returned by the function and any side-effects the function might have are determined only by the function's arguments. If a function is marked as having both noeffects and nostate, the compiler can eliminate redundant calls to the function.

`noeffects` asserts to the compiler that any call to this function will have no effect except to set the return value of the function. If the compiler determines that the return value from a function call is never used, it can remove the call.

`file_scope_vars`(*option*) asserts to the compiler how a function will access variables declared at file scope (with either internal or external linkage).

The *option* is one of the following:

`none` - The function will not read nor write to any file-scope variables except those whose type is `volatile` or those listed in a `#pragma assert global_status_variable`.
`noreads` - The function will not read any file-scope variables except those whose type is `volatile` or those listed in a `#pragma assert global_status_variable`.

`nowrites` - The function will not write to any file-scope variables except those whose type is `volatile` or those listed in a `#pragma assert global_status_variable`.

`format (`*`style`*`, `*`format-index`*`,`*`first-to-check-index`*`)` asserts to the compiler that this function takes `printf`- or `scanf`-style arguments to be type-checked against a format string. Specify the parameters as follows:

style - `printf` or `scanf`.

This determines how the format string is interpreted.

format-index - {1|2|3|...}

This specifies which argument is the format-string argument (starting from 1).

first-to-check-index - {0|1|2|...}

This is the number of the first argument to check against the format string. For functions where the arguments are not available to be checked (such as `vprintf`), specify the third parameter as 0. In this case, the compiler only checks the format string for consistency.

The following declaration causes the compiler to check the arguments in calls to `your_printf` for consistency with the `printf`-style format-string argument `your_format`:

```
extern int
your_printf (void *your_object, const char *your_format, ...);
#pragma assert func_attrs(your_printf) format (printf, 2, 3)
```

The format string (`your_format`) is the second argument of the function `your_printf`, and the arguments to check start with the third argument, so the correct parameter values for *format-index* and *first-to-check-index* are 2 and 3, respectively.

The `format` attribute of `#pragma assert func_attrs` allows you to identify your own functions that take format strings as arguments, so that the compiler can check the calls to these functions for errors. The compiler checks formats for the library functions `printf`, `fprintf`, `sprintf`, `snprintf`, `scanf`, `fscanf`, and `sscanf` whenever these functions are enabled as intrinsics (the default). You can use the `format` attribute to assert that the compiler should check the formats of these functions when they are not enabled as intrinsics.

5.4.1.2. #pragma assert global_status_variable

Use this form of the pragma to specify variables that are to be considered global status variables, which are exempt from any assertions given to functions by `#pragma assert func_attrs file_scope_vars` directives.

The *variable-list* is a list of variables.

5.4.1.3. Usage Notes

The following notes apply to the `#pragma assert func_attrs` and `#pragma assert global_status_variable` forms of the `#pragma assert` directive:

- The `#pragma assert` directive is not subject to macro replacement.
- The variables in the *variable-list* and the identifiers in the *identifier-list* must have declarations that are visible at the point of the `#pragma assert` directive.

- The `#pragma assert` directive must appear at file scope.
- A function can appear on more than one `#pragma assert func_attrs` directive as long as each directive specifies a different assertion about the function. For example, the following is valid:

  ```
  #pragma assert func_attrs(a) nocalls_back
  #pragma assert func_attrs(a) file_scope_vars(noreads)
  ```

  But the following is not valid:

  ```
  #pragma assert func_attrs(a) file_scope_vars(noreads)
  #pragma assert func_attrs(a) file_scope_vars(nowrites)
  ```

5.4.1.4. #pragma assert non_zero

This form of the `#pragma assert` directive is supported on both VAX and Alpha platforms.

When the compiler encounters this directive, it evaluates the *constant-expression*. If the expression is zero, the compiler generates a message that contains both the specified *string-literal* and the compile-time *constant-expression*. For example:

```
#pragma assert non_zero(sizeof(a) == 12) "a is the wrong size"
```

In this example, if the compiler determines that `sizeof a` is not 12, the following diagnostic message is output:

```
CC-W-ASSERTFAIL, The assertion "(sizeof(a) == 12)" was not true.
a is the wrong size.
```

Unlike the `#pragma assert` options `func_attrs` and `global_status_variable`, `#pragma assert non_zero` can appear either inside or outside a function body. When used inside a function body, the pragma can appear wherever a statement can appear, but the pragma is not treated as a statement. When used outside a function body, the pragma can appear anywhere a declaration can appear, but the pragma is not treated as a declaration.

Because macro replacement is not performed on `#pragma assert`, you might need to use the `#pragma assert_m` directive to obtain the results you want. Consider the following program that verifies both the size of a `struct` and the offset of one of its elements:

```
#include <stddef.h>
typedef struct {
    int a;
    int b;
} s;
#pragma assert non_zero(sizeof(s) == 8) "sizeof assert failed"
#pragma assert_m non_zero(offsetof(s,b) == 4) "offsetof assert failed"
```

Because `offsetof` is a macro, the second pragma must be `#pragma assert_m` so that `offsetof` will expand correctly.

5.4.2. #pragma builtins Directive

The `#pragma builtins` directive enables the VSI C built-in functions that directly access processor instructions. This directive is provided for VAX C compatibility.

The `#pragma builtins` directive has the following format:

```
#pragma builtins
```

VSI C implements `#pragma builtins` by including the `<builtins.h>` header file, and is equivalent to `#include <builtins.h>` on OpenVMS systems.

This header file contains prototype declarations for the built-in functions that allow them to be used properly. By contrast, VAX C implemented this pragma with special-case code within the compiler, which also supported a `#pragma nobuiltins` preprocessor directive to turn off the special processing. Because declarations cannot be "undeclared", VSI C does not support `#pragma nobuiltins`.

Furthermore, the names of all the built-in functions use a naming convention defined by the C standard to be in a namespace reserved to the C language implementation. (For more details, see the following Note.)

Note

VAX C implemented both `#pragma builtins` and `#pragma nobuiltins`. Under `#pragma builtins`, the names of the built-in functions were given special treatment. Under `#pragma nobuiltins`, the names of the built-in functions were given no special treatment; as such, a user program was free to declare its own functions or variables with the same names as the builtins and have them behave as if they had ordinary names.

The VSI C implementation relies on the standard C reserved namespace, which states that any name matching the pattern described above is reserved for the exclusive use of the C implementation (that is, the compiler and RTL), and if a user program tries to declare or define such a name for its own purposes, the behavior is undefined.

So in VSI C, the `#pragma builtins` directive includes a set of declarations that makes the built-in functions operate as documented. But in the absence of the `#pragma builtins` directive, you cannot declare your own functions with these names. Code that tries to do anything with these names other than use them as documented, and in the presence of `#pragma builtins`, will likely encounter unexpected problems.

5.4.3. #pragma dictionary Directive

The `#pragma dictionary` directive allows you to extract CDD/Repository data definitions and include these definitions in your program.

The standard-conforming `#pragma dictionary` directive is equivalent to the VAX C compatible `#dictionary` directive (*Section 5.1, "CDD/Repository Extraction (#dictionary)"*), but is supported in all compiler modes. (The `#dictionary` directive is retained for compatibility and is supported only when compiling with the /STANDARD=VAXC qualifier.)

The `#pragma dictionary` directive has the following format:

```
#pragma dictionary CDD_path [null_terminate]
[name (structure_name)]
[text1_to_array | text1_to_char]
```

The *CDD_path* is a character string that gives the path name of a CDD/Repository record, or a macro that expands to the path name of the record.

The optional `null_terminate` keyword can be used to specify that all string data types should be null-terminated.

The optional *name*() can be used to supply an alternate tag name or declarator(*struct_name*) for the outer level of a CDD/Repository structure.

The optional `text1_to_char` keyword forces the CDD/Repository type "text" to be translated to `char`, rather than "array of `char`" if the size is 1. This is the default when `null_terminate` is not specified.

The optional `text1_to_array` keyword forces the CDD/Repository type "text" to be translated to type "array of `char`" even when the size is 1. This is the default when `null_terminate` is specified.

Here's a sample `#pragma dictionary` directive:

```
#pragma dictionary "CDD$TOP.personnel.service.salary_record"
```

This path name describes all subdirectories, beginning with the root directory (CDD$TOP), that lead to the salary_record data definition.

You can use the logical name CDD$DEFAULT to define a default path name for a dictionary directory. This logical name can specify part of the path name for the dictionary object. For example, you can define CDD$DEFAULT as follows:

```
$ DEFINE CDD$DEFAULT CDD$TOP.PERSONNEL
```

When this definition is in effect, the `#pragma dictionary` directive can contain the following:

```
#pragma dictionary "service.salary_record"
```

Descriptions of data definitions are entered into the dictionary in a special-purpose language called CDO (Common Dictionary Operator), which replaces the older interface called CDDL (Common Data Dictionary Language).

CDD definitions written in CDDL are included in a dictionary with the CDDL command. For example, you can write the following definition for a structure containing someone's first and last name:

```
define record cdd$top.doc.cname_record.
   cname structure.
      first    datatype is text
               size is 20 characters.
      last     datatype is text
               size is 20 characters.
   end cname structure.
end cname_record record.
```

If a source file named CNAME.DDL needs to use this definition, you can include the definition in the CDD subdirectory named `doc` by entering the following command:

```
$ CDDL cname
```

After executing this command, a VSI C program can reference this definition with the `#pragma dictionary` directive. If the `#pragma dictionary` directive is not embedded in a VSI C structure declaration, then the resulting structure is declared with a tag name corresponding to the name of the CDD/Repository record. Consider the following example:

```
#pragma dictionary "cdd$top.doc.cname_record"
```

This VSI C preprocessor statement results in the following declarations:

```
struct cname
{
   char first [20];
   char last  [20];
};
```

You can also embed the `#pragma dictionary` directive in another VSI C structure declaration as follows:

```
struct
{
   int id;

#pragma dictionary "cname_record"

}  customer;
```

These lines of code result in the following declaration, which uses `cname` as an identifier for the embedded structure:

```
struct
{
   int id;
   struct
   {
      char first [20];
      char last [20];
   }  cname;
}  customer;
```

If you specify /LIST and either /SHOW=DICTIONARY or /SHOW=ALL in the compilation command line, then the translation of the CDD/Repository record description into VSI C is included in the listing file and marked with the letter D in the margin.

For information on VSI C support for CDD/Repository data types. see *Section C.4.3, "Support for CDD/Repository Data Types"*.

5.4.4. #pragma environment Directive

The `#pragma environment` directive offers a global way to set, save, or restore the states of *context* pragmas. This directive protects include files from contexts set by encompassing programs, and protects encompassing programs from contexts that could be set in header files that they include.

The `#pragma environment` directive affects the following context pragmas:

```
#pragma extern_model
#pragma extern_prefix
#pragma member_alignment
#pragma message
#pragma names
#pragma pointer_size
#pragma required_pointer_size
```

This pragma has the following syntax:

```
#pragma environment command_line
```

```
#pragma environment header_defaults
#pragma environment restore
#pragma environment save
```

The `command_line` keyword sets the states of all the context pragmas as specified on the command line (by default or by explicit use of the /[NO]MEMBER_ALIGNMENT, /[NO]WARNINGS, /EXTERN_MODEL, and /POINTER_SIZE qualifiers). You can use `#pragma environment command_line` within header files to protect them from any context pragmas that take effect before the header file is included.

The `header_defaults` keyword sets the states of all the context pragmas to their default values. This is almost equivalent to the situation in which a program with no command-line options and no pragmas is compiled, except that this pragma sets the pragma message state to `#pragma nostandard`, as is appropriate for header files.

The `save` keyword saves the current state of every pragma that has an associated context.

The `restore` keyword restores the current state of every pragma that has an associated context.

Without requiring further changes to the source code, you can use `#pragma environment` to protect header files from things like language extensions and enhancements that might introduce additional contexts.

A header file can selectively inherit the state of a pragma from the including file and then use additional pragmas as needed to set the compilation to non-default states. For example:

```
#ifdef __pragma_environment
#pragma __environment save  ❶
#pragma __environment header_defaults ❷
#pragma member_alignment restore ❸
#pragma member_alignment save ❹
#endif
.
.  /* contents of header file */
.
#ifdef __pragma_environment
#pragma __environment restore
#endif
```

In this example:

❶ Saves the state of all context pragmas

❷ Sets the default compilation environment

❸ Pops the member alignment context from the `#pragma member_alignment` stack that was pushed by `#pragma __environment save` [restoring the member alignment context to its pre-existing state]

❹ Pushes the member alignment context back onto the stack so that the `#pragma __environment restore` can pop the entry off.

Thus, the header file is protected from all pragmas, except for the member alignment context that the header file was meant to inherit.

5.4.5. #pragma extern_model Directive

The `#pragma extern_model` directive controls how the compiler interprets objects that have external linkage. With this pragma, you can choose one of the following global symbol models to be used for external objects:

- Common block model

  All declarations are definitions, and the linker combines all definitions with the same name into one definition. This is the model traditionally used for `extern` data by VAX C on OpenVMS VAX systems.

- Relaxed ref/def model

  Some declarations are references and some are definitions. Multiple uninitialized definitions for the same object are allowed and resolved into one by the linker. However, a reference requires that at least one definition exists. This model is used by C compilers on UNIX systems.

- Strict ref/def model

  Some declarations are references and some are definitions. There must be exactly one definition in the program for any symbol referenced. This model is the only one guaranteed to be acceptable to all standard C implementations. It is also the one used by VAX C for `globaldef` and `globalref` data. The relaxed ref/def model is the default model on VSI C.

- Globalvalue model

  This is like the strict ref/def model, except that these global objects have no storage; they are, instead, link-time constant values. This model is used by VAX C `globalvalue` symbols.

After a global symbol model is selected with the `extern_model` pragma, all subsequent declarations of objects having external storage class are treated according to the specified model until another `extern_model` pragma is specified.

For example, consider the following pragma:

```
#pragma extern_model strict_refdef
```

After this pragma is specified, the following file-level declarations are treated as declaring global symbols according to the strict ref/def model:

```
int x = 0;
extern int y;
```

Regardless of the external model, the compiler uses standard C rules to determine if a declaration is a definition or a reference, although that distinction is not used in the common block model. An external definition is a file-level declaration that has no storage-class keyword, or that contains the `extern` storage-class keyword, and is also initialized. A reference is a declaration that uses the `extern` storage-class keyword and is not initialized. In the previous example, the declaration of `x` is a global definition and the declaration of `y` is a global reference.

The `extern_model` pragma does not affect the processing of declarations that contain the VAX C keywords `globaldef`, `globalref`, or `globalvalue`.

VSI C also supports the command-line qualifiers /EXTERN_MODEL and /SHARE_GLOBALS to set the external model when the program starts to compile. Pragmas in the program being compiled supersede the command-line qualifier.

A stack of the compiler's external model state is kept so that `#pragma extern_model` can be used transparently in header files and in small regions of program text. See Sections *Section 5.4.5.6, "#pragma extern_model save"* and *Section 5.4.5.7, "#pragma extern_model restore"* for more information.

The compiler issues an error message if the same object has two different external models specified in the same compilation unit, as in the following example:

```
#pragma extern_model common_block
int i = 0;
#pragma extern_model strict_refdef
extern int i;
```

Note

- The global symbols and psect names generated under the control of this pragma obey the case-folding rules of the /NAME qualifier. This behavior is consistent with VAX C.
- While `#pragma extern_model` can be used to allocate several variables in the same psect, the placement of variables relative to each other within that psect cannot be controlled: the compiler does not necessarily allocate distinct variables to memory locations according to the order of appearance in the source code.

  Furthermore, the order of allocation can change as a result of seemingly unrelated changes to the source code, command-line options, or from one version of the compiler to the next; it is essentially unpredictable. The only way to control the placement of variables relative to each other is to make them members of the same `struct` type or, on OpenVMS Alpha systems, by using the `noreorder` attribute on a named `#pragma extern_model strict_refdef`.

See *Section 5.4.5.8, "Effects on the VSI C Run-Time Library and User Programs"* to determine what combinations of external models are compatible for successfully compiling and linking your programs.

The following sections describe the various forms of the `#pragma extern_model` directive.

5.4.5.1. Syntax

The `#pragma extern_model` directive has the following syntax:

```
#pragma extern_model model_spec
[attr[,attr]...]
```

model_spec is one of the following:

```
common_block
relaxed_refdef
strict_refdef "name"
strict_refdef (No attr specifications allowed)
globalvalue (No attr specifications allowed)
```

[*attr*[,*attr*]...] are optional psect attribute specifications chosen from the following (at most one from each line):

```
gbl lcl (Not allowed with relaxed_refdef)
shr noshr
wrt nowrt
pic nopic (Not meaningful for Alpha)
```

```
ovr con
rel abs
exe noexe
vec novec
```
For OpenVMS Alpha systems: `0 byte 1 word 2 long 3 quad 4 octa 5 6 7 8 9 10 11 12 13 14 15 16 page`
For OpenVMS VAX systems: `2 long 3 quad 4 octa 9 page`

The last line of attributes are numeric alignment values. When a numeric alignment value is specified on a section, the section is given an alignment of two raised to that power.

On OpenVMS Alpha and IA-64 systems, the `strict_refdef` "*name*" extern_model can also take the following psect attribute specifications:

- `noreorder` — causes variables in the section to be allocated in the order they are defined.
- `natalgn` — has no effect on OpenVMS systems.

  It does, however, change the behavior on *UNIX* systems: when specified, `natalgn` causes the global variables defined within the section to be allocated on their natural boundary. Currently, all global variables on *UNIX* systems are allocated on a quadword boundary. When the `natalgn` attribute is specified, the compiler instead allocates the variable on an alignment that is natural for its type (`char`s on byte boundaries, `int`s on longword boundaries, and so on).

  Specifying the `natalgn` attribute also enables the `noreorder` attribute.

Note

Use of the `natalgn` attribute can cause a program to violate the *UNIX* Calling Standard. The calling standard states that all global variables must be aligned on a quadword boundary. Therefore, variables declared in a `natalgn` section should only be referenced in the module that defines them.

See *Table 4.3, "Program-Section Attributes"* for a description of the other attributes. See the *OpenVMS Linker Utility Manual* for more complete information on each.

The default attributes are: `noshr`, `rel`, `noexe`, `novec`, `nopic`.

For `strict_refdef`, the default is `con`. For `common_block` and `relaxed_refdef`, the default is `ovr`.

The default for `wrt`/`nowrt` is determined by the first variable placed in the psect. If the variable has the `const` type qualifier (or the `readonly` modifier), the psect is set to `nowrt`. Otherwise, it is set to `wrt`.

Restrictions on Setting Psect Attributes

Be aware of the following restriction on setting psect attributes.

The `#pragma extern_model` directive does not set psect attributes for variables declared as tentative definitions in the `relaxed_refdef model`. A tentative definition is one that does not contain an initializer. For example, consider the following code:

```
#pragma extern_model relaxed_refdef long
int a;
int b = 6;
#pragma extern_model common_block long
```

```
int c;
```

Psect A is given octaword alignment (the default) because `a` is a tentative definition. Psect B is correctly given longword alignment because it is initialized and is, therefore, not a tentative definition. Psect C is also given longword alignment because it is declared in an `extern_model` other than `relaxed_refdef`.

Note

The psect attributes are normally used by system programmers who need to perform declarations normally done in macro. Most of these attributes are not needed in normal C programs. Also, notice that the setting of attributes is supported only through the `#pragma` mechanism, and not through the /EXTERN_MODEL command-line qualifier.

5.4.5.2. #pragma extern_model common_block

This pragma sets the compiler's model of external data to the common block model, which is the one used by VAX C.

The `#pragma extern_model common_block` directive has the following format:

```
#pragma extern_model common_block [attr[,attr]...]
```

In this model, every declaration of an object with the `extern` storage class causes a global overlaid psect to be created. Both standard C definition declarations and reference declarations create the same object file records.

The psect has the same name as the object itself. There is no global symbol in addition to the psect name.

The object file records generated are the same as those generated by VAX C for `extern` objects.

See *Section 4.8, "Program Sections"* for a description of how definitions using each external model are interpreted, what psect they would reside in, and what psect attributes are assigned. Also note the effect of the `const` type specifier for these definitions.

5.4.5.3. #pragma extern_model relaxed_refdef

This pragma sets the compiler's model of external data to the relaxed ref/def model, which is the one used by `pcc` on UNIX systems.

The `#pragma extern_model relaxed_refdef` directive has the following format:

```
#pragma extern_model relaxed_refdef [attr[,attr]...]
```

Be aware that an *attr* keyword of `gbl` or `lcl` is not allowed on the `relaxed_refdef` model.

With this model, three different types of object-file records can be produced, depending on the declaration of the object:

- If the declaration is a standard C reference, the same type of object records are produced as VAX C would produce for a `globalref`; that is, a global symbol reference subrecord.
- If the declaration is a standard C definition that is initialized, a psect definition and global symbol definition subrecord are produced. The name of the psect and symbol is the same as the name of the data object. This is equivalent to what VAX C would produce for the declaration. For example:

```
globaldef "FOO" int FOO = 1;
```

- If the declaration is a standard C definition that is not initialized, then a conditional global symbol definition subrecord and conditional psect definition subrecord are produced. Except for the conditional aspect and the omission of an initializer, these object records resemble those produced with the `#pragma extern_model common_block` directive.

See *Section 4.8, "Program Sections"* for a description of how definitions using each external model are interpreted, what psect they would reside in, and what psect attributes are assigned. Also note the effect of the `const` type specifier for these definitions.

5.4.5.4. #pragma extern_model strict_refdef

This pragma is the preferred alternative to the nonstandard storage-class keywords `globaldef` and `globalref`.

This pragma sets the compiler's model of external data to the strict ref/def model. Use this model for a program that is to be a standard C strictly-conforming program.

The `#pragma extern_model strict_refdef` directive has the following formats:

```
#pragma extern_model strict_refdef
#pragma extern_model strict_refdef "name" [attr[,attr]...]
```

The name in quotes, if specified, is the name of the psect for any definitions.

Note that *attr* keywords cannot be specified for the `strict_refdef` model unless a name is given for the psect.

This model provides two different cases:

- If the declaration is a standard C reference, the same type of object records are produced as VAX C would produce for a `globalref`; that is, a global symbol reference subrecord.
- If the declaration is a standard C definition, the same type of object records are produced as VAX C would produce for a `globaldef`; that is, a global symbol definition subrecord.

See *Section 4.8, "Program Sections"* for a description of how definitions using each external model are interpreted, what psect they would reside in, and what psect attributes are assigned. Also note the effect of the `const` type specifier for these definitions.

Note

In VAX C, the `globaldef` and `globalref` keywords interact with `enum` definitions in the following way:

- If an `enum` variable is declared with the `globaldef` keyword, the `enum` literals of the type of the variable automatically become `globalvalue` constant definitions.
- If an `enum` variable is declared with the `globalref` keyword, the `enum` literals of the type of the variable automatically become `globalvalue` constant references.

This behavior, does not occur with `#pragma extern_model strict_refdef`.

5.4.5.5. #pragma extern_model globalvalue

This pragma sets the compiler's external model to the `globalvalue` model, and is the preferred alternative to the nonstandard storage-class keyword `globalvalue`.

This pragma has the following format:

```
#pragma extern_model globalvalue
```

Notice that this model does not accept *attr* keywords.

This model provides two different cases:

- If the declaration is a standard C reference, the same object file records are produced as VAX C would produce for an uninitialized `globalvalue`.
- If the declaration is a standard C definition, the same object records are produced as VAX C would produce for an initialized `globalvalue`.

Note

Only objects with a type of `integer`, `enum`, or `pointer` can have this external model. If this external model is used and the compiler encounters a declaration of an external object whose type is not one these, an error message is issued.

5.4.5.6. #pragma extern_model save

This pragma pushes the current external model of the compiler onto a stack. The stack records all information associated with the external model, including the `shr`/`noshr` state and any quoted psect name.

This pragma has the following format:

```
#pragma extern_model save
```

The number of entries allowed in the `#pragma extern_model` stack is limited only by the amount of memory available to the compiler.

5.4.5.7. #pragma extern_model restore

This pragma pops the external model stack of the compiler. The external model is set to the state popped off the stack. The stack records all information associated with the external model, including the `shr`/`noshr` state and any quoted psect name. This pragma has the following format:

```
#pragma extern_model restore
```

On an attempt to pop an empty stack, a warning message is issued and the compiler's external model is not changed.

5.4.5.8. Effects on the VSI C Run-Time Library and User Programs

Using different VSI C external models can introduce mutually incompatible object files. An object file compiled with one extern model may not link against an object file compiled with a different model.

Table 5.1, "Comparison of Mixing Different extern_models" compares what happens when a reference or definition in an object file compiled with one external model is linked against a reference or definition

in an object file compiled with a different external model. Note that the table is symmetric about the diagonal. For example, to look up what happens when you mix a `relaxed_refdef` reference with a `strict_refdef` definition, you can locate either the `relaxed_refdef` reference row and the `strict_refdef` definition column or the `relaxed_refdef` reference column and the `strict_refdef` definition row.

Table 5.1, "Comparison of Mixing Different extern_models" contains no entries for mixing `globalvalue` symbols with other external models because `globalvalue` symbols are used only in special cases; they are not used as a general-purpose external model. For the other external models, there is a row and column for every different case. The `common_block` model only has one case because all symbols are definitions in that model; the `relaxed_refdef` model has three cases because it distinguishes between references, uninitialized definitions, and initialized definitions.

Table 5.1. Comparison of Mixing Different extern_models

	common_ block def	**relaxed_ refdef ref**	**relaxed_ refdef def**	**relaxed_ refdef initialized def**	**strict_ refdef ref**	**strict_ refdef def**
common_block def	Works	Fails	Works	Works	Fails	Fails
relaxed_refdef ref	Fails	Works	Works	Works	Works	Works
relaxed_refdef uninitialized def	Works	Works	Works	Works	Works	Works
relaxed_refdef initialized def	Works	Works	Works	Multi	Works	Multi
strict_refdef ref	Fails	Works	Works	Works	Works	Works
strict_refdef def	Fails	Works	Works	Multi	Works	Multi

Notes

ref means reference; def means definition.
In the `common_block` model, all external symbols are considered to be defs.
A ref works with a ref if they both refer to the same thing.
A def works with a ref if the def fulfills the ref.
A def works with a def if they are combined into one by the linker.

Multi means that the linker issues a multiply defined symbol error. This indicates a user error, not a mismatch between external models.

As *Table 5.1, "Comparison of Mixing Different extern_models"* shows, the `common_block` model mixes poorly with the `strict_refdef` model, but the `relaxed_refdef` model works well with the `common_block` model and the `strict_refdef` model. The `relaxed_refdef` model fails only when a `relaxed_refdef` reference is linked against a `common_block` definition.

The fact that the external models are not all compatible with each other can be an issue for providers of general-purpose object libraries. One goal for such a library should be to work when linked with client code compiled with any of the external models. Otherwise, the provider of the object library might be forced to provide one copy of the library compiled with /EXTERN_MODEL=COMMON_BLOCK, another compiled with /EXTERN_MODEL=STRICT_REFDEF, and another compiled with /EXTERN_MODEL=RELAXED_REFDEF to let anyone link with the library.

The best way to accomplish the goal of allowing an object library to be linked with any code regardless of the external model used, is to provide header files that describe the interface to the object library. The header files can declare the global variables used by the object library after using `#pragma extern_model` to set the external model to the one used by the library. Programmers who want to use the library could then include these header files to get the required declarations. In order to avoid altering the external model used by the including program, header files should start with a `#pragma extern_model save` directive and end with a `#pragma extern_model restore` directive. The VSI C RTL uses this approach.

If header files are not provided, an object library should use the `relaxed_refdef` external model since it will link successfully with either `common_block` compiled code or `strict_refdef` compiled code. The only restriction is that the library must not reference an external symbol that is not defined in the library but is defined only in the user program. This avoids the `common_block` case that fails. Note that the `relaxed_refdef` model allows both the library and the user code to contain definitions for any symbol, as long as both do not attempt to initialize the symbol.

5.4.5.9. Example

Example 5.1, "#pragma extern_model Example" shows the use of `#pragma extern_model` in a sample module. Assume that the module is compiled with the /EXTERN_MODEL=COMMON and /SHARE_GLOBALS qualifiers.

Example 5.1. #pragma extern_model Example

```
#pragma extern_model save
❶globaldef {"BAR1"} int FOO1;          /* strict_refdef shr def  */
❷extern int com1;                      /* common_block shr def   */
❸int com2;                             /* common_block shr def   */
#pragma extern_model common_block noshr
❹globaldef {"BAR2"} int FOO2;          /* strict_refdef shr def  */
❺extern int com3 = 23;                 /* common_block noshr def */
#pragma extern_model globalvalue
❻int gv1;                              /* globalvalue def        */
❼extern int gv2;                       /* globalvalue ref        */
❽int gv3 = 5;                          /* globalvalue def        */
❾extern int gv4 = 42;                  /* globalvalue def        */
#pragma extern_model strict_refdef {"BAR1"} shr
❿int FOO1A;                            /* strict_refdef shr def  */
⓫extern int FOO1B;                     /* strict_refdef ref      */
⓬globaldef {"BAR3"} noshare int foo3;
#pragma extern_model relaxed_refdef
⓭int rrd1;                             /* relaxed_refdef noshr def */
⓮extern rrd2;                          /* relaxed_refdef ref     */
#pragma extern_model restore
⓯int com4;                             /* common_block shr def   */
```

Key to *Example 5.1, "#pragma extern_model Example"*:

❶ `FOO1` has the `strict_refdef` model with the share attribute (because of /SHARE). It resides in psect `BAR1`.

❷ `com1` has the `common_block` model with the share attribute. Like all common_block globals, `com1` is a definition.

❸ `com2` has the `common_block` model with the share attribute. Like all common_block globals, `com2` is a definition.

❹ FOO2 has the strict_refdef model with the share attribute. The /SHARE qualifier overrides the noshr keyword on the preceding #pragma extern_model. FOO2 resides in psect BAR2.

❺ com3 has the common_block model with the noshare attribute.

❻ gv1 has the globalvalue model. It is a definition. Since it lacks an explicit initializer, gv1 is implicitly initialized to 0. Therefore, it is a globalvalue with a link-time value of 0.

❼ gv2 has the globalvalue model. It is a reference.

❽ gv3 has the globalvalue model. It is a definition with a link-time value of 5.

❾ gv4 has the globalvalue model. It is a definition with a link-time value of 42.

❿ FOO1A has the strict_refdef model with the noshare attribute. It is a definition and resides in the psect BAR1.

⓫ FOO1B has the strict_refdef model and is a reference. Since it is a reference, it will reside in whatever psect is specified by the definition.

⓬ foo3 has the strict_refdef model with the noshare attribute. It is a definition and resides in the psect BAR3.

⓭ rrd1 has the relaxed_refdef model with the noshare attribute. It is a definition.

⓮ rrd2 has the relaxed_refdef model and is a reference.

⓯ com4 has the common_block model with the share attribute, because the preceding line popped the external model back to its command-line state.

5.4.6. #pragma extern_prefix Directive

The #pragma extern_prefix directive controls the compiler's synthesis of external names, which the linker uses to resolve external name requests.

When you specify #pragma extern_prefix with a string argument, the compiler attaches the string to the beginning of all external names produced by the declarations that follow the pragma specification.

This pragma is useful for creating libraries where the facility code can be attached to the external names in the library.

The #pragma extern_prefix directive has the following format:

```
#pragma extern_prefix "string" [(id[,id]...)]
#pragma extern_prefix {NOCRTL|RESTORE_CRTL} (id[,id]...)
#pragma extern_prefix  save
#pragma extern_prefix  restore
```

The quoted *"string"* is attached to external names in the declarations that follow the pragma specification.

You can also specify an extern prefix for specific identifiers using the optional list [(*id*[,*id*]...)].

The NOCRTL and RESTORE_CRTL keywords control whether or not the compiler applies its default RTL prefixing to the names specified in the *id*-list, which is required for this form of the pragma.

The effect of `NOCRTL` is like that of the EXCEPT=*keyword* of the /PREFIX_LIBRARY_ENTRIES command-line qualifier. The effect of `RESTORE_CRTL` is to undo the effect of a `#pragma extern_prefix NOCRTL` or a /PREFIX=EXCEPT= on the command line.

The `save` and `restore` keywords can be used to save the current pragma prefix string and to restore the previously saved pragma prefix string, respectively.

The default external prefix, when none has been specified by a pragma, is the null string.

The recommended use is as follows:

```
#pragma extern_prefix save
#pragma extern_prefix "prefix-to-prepend-to-external-names"
… some declarations and definitions …
#pragma extern_prefix restore
```

When an `extern_prefix` is in effect and you are using `#include` to include header files, but do not want the `extern_prefix` to apply to `extern` declarations in the header files, use the following code sequence:

```
#pragma extern_prefix save
#pragma extern_prefix ""
#include ...
#pragma extern_prefix restore
```

Otherwise, external prefix is attached to the beginning of external identifiers for definitions in the included files.

All external names prefixed with a nonnull string using `#pragma extern_prefix` are converted to uppercase letters, regardless of the setting of the /NAMES qualifier.

Note

The following notes apply when specifying optional identifiers on `#pragma extern_prefix`:

- When an *id*-list follows a quoted *"string"*, then for each *id* there must not be a declaration of that *id* visible at the point of the pragma, otherwise a warning is issued, and there is no affect on that *id*.
- Each *id* affected by a pragma with a non-empty prefix is expected to be subsequently declared with external linkage in the same compilation unit. The compiler issues a default informational if there is no such declaration made by the end of the compilation.
- It is perfectly acceptable for the *id*-list form of the pragma or declarations of the *id*'s listed, to occur within a region of source code controlled by the other form of the pragma. The two forms do not interact; the form with an *id* list always supersedes the other form.
- There is no interaction between the save/restore stack and the *id* lists.
- If the same *id* appears in more than one pragma, then a default informational message is issued, unless the prefix on the second pragma is either empty ("") or matches the prefix from the previous pragma. In any case, the behavior is that the last-encountered prefix supersedes all others.

5.4.7. #pragma function Directive

Specifies that calls to the specified functions are not intrinsic but are, in fact, function calls. This pragma has the opposite effect of `#pragma intrinsic`.

The `#pragma function` directive has the following format:

```
#pragma function (function1[, function2, ...])
```

5.4.8. #pragma [no]include_directory Directive

The effect of each `#pragma include_directory` is as if its string argument (including the quotes) were appended to the list of places to search that is given its initial value by the /INCLUDE_DIRECTORY qualifier, except that an empty string is not permitted in the pragma form.

The `#pragma include_directory` directive has the following format:

```
#pragma include_directory <string-literal>
```

This pragma is intended to ease DCL command-line length limitations when porting applications from POSIX-like environments built with makefiles containing long lists of -I options specifying directories to search for headers. Just as long lists of macro definitions specified by the /DEFINE qualifier can be converted to `#define` directives in a source file, long lists of places to search specified by the /INCLUDE_DIRECTORY qualifier can be converted to `#pragma include_directory` directives in a source file.

Note that the places to search, as described in the help text for the /INCLUDE_DIRECTORY qualifier, include the use of POSIX-style pathnames, for example `"/usr/base"`. This form can be very useful when compiling code that contains POSIX-style relative pathnames in `#include` directives. For example, `#include <subdir/foo.h>` can be combined with a place to search such as `"/usr/base"` to form `"/usr/base/subdir/foo.h"`, which will be translated to the filespec "USR:[BASE.SUBDIR]FOO.H"

This pragma can appear only in the main source file or in the first file specified on the /FIRST_INCLUDE qualifier. Also, it must appear before any `#include` directives.

5.4.9. #pragma [no]inline Directive

Function inlining is the inline expansion of function calls; it replaces the function call with the function code itself. Inline expansion of functions reduces execution time by eliminating function-call overhead and allowing the compiler's general optimization methods to apply across the expanded code. Compared with the use of function-like macros, function inlining has the following advantages:

- Arguments are evaluated only once.
- The overuse of parentheses is not necessary to avoid problems with precedence.
- The actual expansion can be controlled from the command line.

Also, the semantics are exactly the same as if inline expansion had not occurred. You cannot get this behavior using macros.

Use the following preprocessor directives to control function inlining:

```
#pragma inline ( id,… )

#pragma noinline ( id,… )
```

The *id* is a function identifier.

If a function is named in an `inline` directive, calls to that function will be expanded as inline code, if possible.

If a function is named in a `noinline` directive, calls to that function will *not* be expanded as inline code.

If a function is named in both an `inline` and a `noinline` directive, an error message is issued.

For calls to functions named in neither an `inline` nor a `noinline` directive, VSI C expands the function as inline code whenever appropriate as determined by a platform-specific algorithm.

Use of the `#pragma inline` directive causes inline expansion, regardless of the size or number of times the specified functions are called.

In the following example of function inlining, the functions `push` and `pop` are expanded inline throughout the module in which the `#pragma inline` appears:

```
void push(int);
int pop(void);

#pragma inline(push, pop)

int stack[100];
int *stackp = &stack;

void push(int x)
{
    if (stackp == &stack)
        *stackp = x;
    else
        *stackp++ = x;
}

int pop()
{
    return *stackp-;
}

main()
{

  push(1);
  printf("The top of stack is now %d \n",pop());
}
```

By default, VSI C for OpenVMS systems attempts to provide inline expansion for all functions, and uses the following function characteristics to determine if it can provide inline expansion:

- Size
- Number of times the function is called
- Conformance to the following restrictions:
    - The function does not take the address of a parameter.
    - The function does not use an index expression that is not a compile-time constant in an array that is a field of a `struct` argument. An argument that is a pointer to a `struct` is not restricted.

- The function does not use the `varargs` or `stdarg` package to access the function's arguments because they require arguments to be in adjacent memory locations, and inline expansion may violate that requirement.
- The function does not declare an exception handler.

If a function is to be expanded inline, you must place the function definition in the same module as the function call. The definition can appear either before or after the function call.

5.4.10. #pragma intrinsic Directive

The `#pragma intrinsic` preprocessor directive specifies that calls to the specified functions are intrinsic. An intrinsic function is an apparent function call that could be handled as an actual call to the specified function, or could be handled by the compiler in a different manner. By treating the function as an intrinsic, the compiler can often generate faster code. (Contrast with a built-in function, which is an apparent function call that is never handled as an actual function call. There is never a function with the specified name.)

This pragma has the opposite effect of `#pragma function`.

The `#pragma intrinsic` directive has the following format:

```
#pragma intrinsic (function1[,function2, …])
```

Functions that can be handled as intrinsics are:

```
Main Group - Standard C:

  abs     atan2   ceilf   cosl    floorl   memset  sinl
  atan    atan2f  ceill   fabs    labs     sin     strcpy
  atanf   atan2l  cos     floor   memcpy   sinf    strlen
  atanl   ceil    cosf    floorf  memmove

Main Group - Nonstandard:

  alloca  atand   atand2  bcopy   bzero    cosd    sind

Printf functions:

  fprintf printf sprintf

Printf Nonstandard:

  snprintf

Standard math functions that set errno,

thereby requiring /ASSUME=NOMATH_ERRNO:

  acos    asinl   expf    log10    powl   sqrtf  tanh
  acosf   cosh    expl    log10f   sinh   sqrtl  tanhf
  acosl   coshf   log     log10l   sinhf  tan    tanhl
  asin    coshl   logf    pow      sinhl  tanf
  asinf   exp     logl    powf     sqrt   tanl

Nonstandard math functions that set errno,
```

```
thereby requiring /ASSUME=NOMATH_ERRNO:

  log2
  tand
```

Also see *Section 1.3.4, "CC Command Qualifiers"* for a description of the [NO]INTRINSICS option of the /OPTIMIZE qualifier, which controls whether or not certain functions are handled as intrinsic functions without explicitly enabling each of them as an intrinsic through the `#pragma intrinsic` directive.

Also, the `asm`, `fasm`, and `dasm` functions are intrinsics and require use of `#pragma intrinsic`. See *Section 6.2.1.2, "In-line Assembly Code – ASMs"* for a description of these functions.

5.4.11. #pragma linkage Directive (Alpha only)

This section describes the behavior of the `#pragma linkage` directive on OpenVMS Alpha systems.

The `#pragma linkage` preprocessor directive allows you to specify special linkage types for function calls. This pragma is used with the `#pragma use_linkage` directive, described in *Section 5.4.23, "#pragma use_linkage Directive "*, to associate a previously defined special linkage with a function.

For OpenVMS Alpha systems, the `#pragma linkage` directive has the following formats:

```
#pragma linkage linkage-name = (characteristics)
#pragma linkage_alpha linkage-name = (characteristics)
```

Both formats behave identically on OpenVMS Alpha systems. On IA-64 systems, however, register mapping occurs for the `pragma linkage` format, as described in *Section 5.4.12, "#pragma linkage Directive* (I64 only)*"*.

The *linkage-name* is the name to be given to the linkage type being defined. It has the form of a C identifier. Linkage types have their own name space, so their names will not conflict with other identifiers or keywords in the compilation unit.

The *characteristics* specify information about where parameters will be passed, where the results of the function are to be received, and what registers are modified by the function call. Specify these *characteristics* as a parenthesized list of comma-separated items of the following forms:

```
parameters (register-list)
result     (simple-register-list)
preserved  (simple-register-list)
nopreserve (simple-register-list)
notused    (simple-register-list)
notneeded  (ai, lp)
standard_linkage
```

If the `standard_linkage` keyword is specified, it must be the only option in the parenthesized list following the linkage name. For example:

```
#pragma linkage special1 = (standard_linkage)
```

The `standard_linkage` keyword tells the compiler to use the standard linkage appropriate to the target platform. This can be useful to confine conditional compilation to the pragmas that define linkages, without requiring the corresponding `#pragma use_linkage` directives to be conditionally compiled as well.

Code written to use linkage pragmas as intended, treating them as target-specific without implicit mapping, might have a form like this:

```
#if defined(__alpha)
#pragma linkage_alpha special1 = (__preserved(__r1,__r2))
#elif defined(__ia64)
#pragma linkage_ia64 special1 = (__preserved(__r9,__r28))
#else
#pragma message ("unknown target, assuming standard linkage")
#pragma linkage special1 = (standard_linkage)
#endif
```

If the `standard_linkage` keyword is not specified, you can supply the `parameters`, `result`, `preserved`, `nopreserve`, `notused`, and `notneeded` keywords in any order.

A *simple-register-list* is a comma-separated list of register names, either Rn or Fn, where n is a valid register number. A *register-list* is similar to a *simple-register-list* except that it can contain parenthesized sublists.

For OpenVMS Alpha systems, valid registers for the `preserved`, `nopreserve`, and `notused` options are:

- General-purpose registers R0 through R30
- Floating-point registers F0 through F30

Valid registers for the `result` and `parameters` options are:

- General-purpose registers R0 through R25
- Floating-point registers F0 through F30

For example, the following characteristics specify a *simple-register-list* containing two elements, registers F3 and F4; and a *register-list* containing two elements, the register R5 and a sublist containing the registers F5 and F6:

```
nopreserve(f3, f4)
parameters(r5, (f5, f6))
```

The following example shows a linkage using such characteristics:

```
#pragma linkage my_link=(nopreserve(f3,f4), parameters(r5,(f5,f6)),
 notneeded (ai))
```

The parenthesized notation in a *register-list* is used to describe arguments and function return values of type `struct`, where each member of the `struct` is passed in a single register. In the following example, `sample_linkage` specifies two parameters: the first is passed in registers R5, R6, and R7; the second is passed in F6:

```
struct sample_struct_t {
    int A, B;
    short C;
    } sample_struct;

#pragma linkage sample_linkage = (parameters ((r5, r6, r7), f6))
void sub (struct sample_struct_t p1, double p2) { }
```

```
main()
{
    double d;

    sub (sample_struct, d);
}
```

You can pass arguments to the parameters of a routine in specific registers. To specify this information, use the following form, where each item in the *register-list* describes one parameter that is passed to the routine:

```
parameters (register-list)
```

You can pass structure arguments by value, with the restriction that each member of the structure is passed in a separate parameter location. Doing so, however, may produce code that is slower because of the large number of registers used. The compiler does not diagnose this condition.

VSI C does not support unions as parameters or function return types for a function with a special linkage.

When a function associated with a linkage type is declared or defined, the compiler checks that the size of any declared parameters is compatible with the number of registers specified for the corresponding parameter in the linkage definition.

The compiler needs to know the registers that will be used to return the value for the function. To specify this information use the following form, where the *register-list* must contain only a single register, or a parenthesized group of registers if the routine returns a `struct`:

```
result (register-list)
```

If a function does not return a value (that is, the function has a return type of `void`), then do not specify `result` as part of the linkage.

The compiler needs to know which registers are used by the function and which are not, and of those used, whether or not they are preserved across the function call. To specify this information, use the following forms:

```
preserved (register-list)
nopreserve (register-list)
notused (register-list)
```

A `preserved` register contains the same value after a call to the function as it did before the call.

A `nopreserve` register does not necessarily contain the same value after a call to the function as it did before the call.

A `notused` register is not used in any way by the called function.

The `notneeded` characteristic indicates that certain items are not needed by the routines using this linkage. You can specify one or both of the following keywords:

- `ai` – Specifies that the Argument Information register (R25) does not need to be set up when calling the specified functions.
- `lp` – Specifies that the Linkage Pointer register (R27 for Alpha systems) does not need to be set up when calling the specified functions. The linkage pointer is required when the called function

accesses global or `static` data. For IA-64 systems, there is no linkage pointer, so this setting is accepted but does not change the behavior of the pragma.

You must determine whether or not it is valid to specify that the `ai` or `lp` registers are not needed.

The `#pragma linkage` directive has the restriction that structures containing nested substructures are not supported as parameters or function return types with special linkages. Also, functions that use the __RETURN_ADDRESS built-in function or `va_count` C RTL function cannot be called with a special linkage.

5.4.12. #pragma linkage Directive (I64 only)

The `#pragma linkage` directive behaves much the same on IA-64 systems as it does on OpenVMS Alpha systems, with some important differences.

on IA-64 systems, the `#pragma linkage` directive has the following formats:

```
#pragma linkage linkage-name = (characteristics)
#pragma linkage_ia64 linkage-name = (characteristics)
```

5.4.12.1. #pragma linkage Format

on IA-64 systems, the `#pragma linkage` format of this directive accepts Alpha register names and conventions and automatically maps them, where possible, to specific I64 registers. So whenever VSI C for I64 encounters a `#pragma linkage` directive, it attempts to map the Alpha registers specified in the linkage to corresponding I64 registers, and emits a SHOWMAPLINKAGE informational message showing the I64 specific form of the directive, `#pragma linkage_ia64`, with the I64 register names that replaced the Alpha register names. The SHOWMAPLINKAGE message is suppressed under the `#pragma nostandard` directive, normally used within system header files.

Code compiled on IA-64 systems that deliberately relies on the register mapping performed by `#pragma linkage` should either ignore the SHOWMAPLINKAGE informational, or disable it.

5.4.12.1.1. Register Mapping

Table 5.2, "Integer Register Mapping" shows the mapping that VSI C applies to the Alpha integer register names used in `#pragma linkage` directives when they are encountered on an IA-64 system. Note that the six standard parameter registers on Alpha (R16-R21) are mapped to the first six (of eight) standard parameter registers on IA-64 systems, which happen to be stacked registers (see *Section 5.4.12.2, "#pragma linkage_ia64 Format"*).

Table 5.2. Integer Register Mapping

Alpha →	**I64**	**Alpha →**	**I64**
R0	R8	R16	R32[1]
R1	R9	R17	R33[1]
R2	R28	R18	R34[1]
R3	R3	R19	R35[1]
R4	R4	R20	R36[1]
R5	R5	R21	R37[1]
R6	R6	R22	R22

Alpha →	I64	Alpha →	I64
R7	R7	R23	R23
R8	R26	R24	R24
R9	R27	R25	R25
R10	R10	R26	no mapping
R11	R11	R27	no mapping
R12	R30	R28	no mapping
R13	R31	R29	R29
R14	R20	R30	R12
R15	R21	R31	R0

[1]In parameters or result; else ignored

Table 5.3, "Floating-Point Register Mapping" shows the mapping that VSI C applies to the Alpha floating-point register names used in `#pragma linkage` directives when they are encountered on an IA-64 system:

Table 5.3. Floating-Point Register Mapping

Alpha →	I64	Alpha →	I64
F0	F8	F16	F8
F1	F9	F17	F9
F2	F2	F18	F10
F3	F3	F19	F11
F4	F4	F20	F12
F5	F5	F21	F13
F6	F16	F22	F22
F7	F17	F23	F23
F8	F18	F24	F24
F9	F19	F25	F25
F10	F6	F26	26
F11	F7	F27	27
F12	F20	F28	28
F13	F21	F29	F29
F14	F14	F30	F30
F15	F15		

5.4.12.1.2. Mapping Diagnostics

In some cases, the VSI C compiler on Alpha systems silently ignores linkage registers if, for example, a standard parameter register like R16 is specified in a `preserved` option. When you compile on an IA-64 system, this situation emits an MAPREGIGNORED informational message, and the

SHOWMAPLINKAGE output might not be correct. If there is no valid mapping to I64 registers, the NOMAPPOSSIBLE error message is output. There are two special situations that can arise when floating-point registers are specified in a linkage:

- Only IEEE-format values are passed in floating-point registers under the OpenVMS Calling Standard for I64: VAX format values are passed in integer registers. Therefore, a compilation that specifies /FLOAT=D_FLOAT or /FLOAT=G_FLOAT produces an error for any linkage that specifies floating-point registers. Note that this includes use in options that do not involve passing values, such as the `preserved` and `notused` options.
- The mapping of floating-point registers is many-to-one in two cases:
  - Alpha registers F0 and F16 both map to I64 register F8
  - Alpha F1 and F17 both map to I64 register F9.

  A valid Alpha linkage may well specify uses for both F0 and F16, and/or both F1 and F17. Such a linkage cannot be mapped on an IA-64 system. But because of the way this situation is detected, the MULTILINKREG warning message that is produced can only identify the second occurrence of an Alpha register that got mapped to the same I64 register as some previous Alpha register. The actual pair of Alpha registers in the source is not identified, and so the message can be confusing. For example, an option like `preserved(F1,F17)` gets a MULTILINKREG diagnostic saying that F17 was specified more than once.

5.4.12.2. #pragma linkage_ia64 Format

The `#pragma linkage_ia64` format requires register names to be specified in terms of an IA-64 system. The register names will never be mapped to a different architecture. This form of the pragma always produces an error if encountered on a different architecture.

For this format of the pragma, valid registers for the `preserved`, `nopreserve`, `notused`, `parameters`, and `result` options are:

- Integer registers R3 through R12 and R19 through R31
- Floating-point registers F2 through F31

Valid registers for the `parameters` and `result` are:

- Integer registers R3 through R12, and R19 through R31
- Integer registers R32 through R39 (according to the convention described below)
- Floating-point registers F2 through F31

The `parameters` and `result` options permit integer registers R32 through R39 to be specified according to the following convention: On IA-64, the first eight integer input/output slots are allocated to stacked registers, and thus the calling routine refers to them using different names than the called routine. The convention for naming these registers in either the `parameters` or `result` option of a `#pragma linkage_ia64` directive is always to use the hardware names as they would be used within the *called* routine: R32 through R39. The compiler automatically compensates for the fact that within the calling routine these same registers are designated using different hardware names.

5.4.13. #pragma [no]member_alignment Directive

By default, VSI C for OpenVMS VAX systems does not align structure members on natural boundaries; they are stored on byte boundaries (with the exception of bit-field members).

By default, VSI C for OpenVMS Alpha systems does align structure members on natural boundaries.

The `#pragma member_alignment` preprocessor directive can be used to force natural-boundary alignment of structure members. The `#pragma nomember_alignment` preprocessor directive restores byte-alignment of structure members.

This pragma has the following formats:

```
#pragma member_alignment
#pragma member_alignment save
#pragma member_alignment restore
#pragma nomember_alignment [base_alignment]
```

When `#pragma member_alignment` is used, the compiler aligns structure members on the next boundary appropriate to the type of the member, rather than on the next byte. For example, a `long` variable is aligned on the next longword boundary; a `short` variable is aligned on the next word boundary.

Consider the following example:

```
#pragma nomember_alignment

struct x {
           char c;
           int b;
           };

#pragma member_alignment

struct y {
          char c;       /*3 bytes of filler follow c */
          int b;
          };

main ()

{
        printf( "The sizeof y is: %d\n", sizeof (struct y) );
        printf( "The sizeof x is: %d\n", sizeof (struct x) );
}
```

When this example is executed, it shows the difference between `#pragma member_alignment` and `#pragma nomember_alignment`.

Once used, the `member_alignment` pragma remains in effect until the `nomember_alignment` pragma is encountered; the reverse is also true.

The optional *base_alignment* parameter can be used to specify the base-alignment of the structure. Use one of the following keywords for the *base_alignment*:

- `byte` (1 byte)
- `word` (2 bytes)
- `longword` (4 bytes)

- `quadword` (8 bytes)
- `octaword` (16 bytes)

The `#pragma member_alignment save` and `#pragma member_alignment restore` directives can be used to save the current state of the `member_alignment` and to restore the previous state, respectively. This feature is necessary for writing header files that require `member_alignment` or `nomember_alignment`, or that require inclusion in a `member_alignment` that is already set.

5.4.14. #pragma message Directive

The `#pragma message` directive controls the issuance of individual diagnostic messages or groups of messages. Use of this pragma overrides any command-line options that may affect the issuance of messages.

The `#pragma message` directive has the following formats:

```
#pragma message option1 (message-list)
#pragma message option2
#pragma message (quoted-string)
```

5.4.14.1. #pragma message *option1*

The parameter *option1* must be one of the following keywords:

- `enable` — Enables issuance of the messages specified in the *message-list*
- `disable` — Disables issuance of the messages specified in the *message-list*
- `emit_once` — Emits the specified messages only once per compilation.

  Certain messages are emitted only the first time the compiler encounters the causal condition. When the compiler encounters the same condition later in the program, no message is emitted. Messages about the use of language extensions are an example of this kind of message. To emit one of these messages every time the causal condition is encountered, use the EMIT_ALWAYS option.

  Errors and Fatals are always emitted. You cannot set them to `emit_once`.
- `emit_always` — Emits the specified messages at every occurrence of the condition.
- `error` — Sets the severity of the specified messages to Error.

  Supplied Error messages and Fatal messages cannot be made less severe. (Exception: A message can be upgraded from Error to Fatal, then later downgraded to Error again, but it can never be downgraded from Error.)

  Warnings and Informationals can be made any severity.)
- `fatal` — Sets the severity of the specified messages to Fatal.
- `informational` — Sets the severity of the specified messages to Informational. Note that Fatal and Error messages cannot be made less severe.
- `warning` — Sets the severity of each message in the message-list to Warning. Note that Fatal and Error messages cannot be made less severe.

The *message-list* can be any one of the following:

- A single message identifier (within parentheses, or not). The message identifier is the name following the severity at the start of a line when a message is issued. For example, in the following message, the message identifier is GLOBALEXT:

  ```
  %CC-W-GLOBALEXT, a storage class of globaldef, globalref, or globalvalue
  is a language extension.
  ```

- The name of a single message group (within parentheses, or not). Message-group names are:

  - ALL – All the messages in the compiler
  - ALIGNMENT – Messages about unusual or inefficient data alignment.
  - C_TO_CXX – Messages reporting the use of C features that would be invalid or have a different meaning if compiled by a C++ compiler.
  - CDD – Messages about CDD (Common Data Dictionary) support.
  - CHECK – Messages reporting code or practices that, although correct and perhaps portable, are sometimes considered ill-advised because they can be confusing or fragile to maintain. For example, assignment as the test expression in an "if" statement.

    The check group gets defined by enabling LEVEL5 messages.
  - DEFUNCT – Messages reporting the use of obsolete features: ones that were commonly accepted by early C compilers but were subsequently removed from the language.
  - NEWC99 – Messages reporting the use of the new C99 Standard features.
  - NOANSI – This is a deprecated message group. It is an obsolete synonym for NOC89. Also see message groups NEWC99, NOC89, NOC99.
  - NOC89 – Messages reporting the use of non-C89 Standard features.
  - NOC99 – Messages reporting the use of non-C99 Standard features.
  - OBSOLESCENT – Messages reporting the use of features that are valid in Standard C, but which were identified in the standard as being obsolescent and likely to be removed from the language in a future version of the standard.
  - OVERFLOW – Messages that report assignments and/or casts that can cause overflow or other loss of data significance.
  - PERFORMANCE – Messages reporting code that might result in poor run-time performance.
  - PORTABLE – Messages reporting the use of language extensions or other constructs that might not be portable to other compilers or platforms.
  - PREPROCESSOR – Messages reporting questionable or non-portable use of preprocessing constructs.
  - QUESTCODE – Messages reporting questionable coding practices. Similar to the CHECK group, but messages in this group are more likely to indicate a programming error rather than just a non-robust style.

Note

Enabling the QUESTCODE group provides lint-like checking.

- RETURNCHECKS – Messages related to function return values.
- UNINIT – Messages related to using uninitialized variables.
- UNUSED – Messages reporting expressions, declarations, header files, CDD records, static functions, and code paths that are not used.

  Note, however, that unlike any other messages, these messages must be enabled on the command line (/WARNINGS=ENABLE=UNUSED) to be effective.

- A single message-level name (within parentheses, or not).

  Message-level names are:

  - LEVEL1 – Important messages. These are less important than the level 0 core messages, because messages in this group are not displayed if `#pragma nostandard` is active.
  - LEVEL2 – Moderately important messages.
  - LEVEL3 – Less important messages.

    LEVEL3 is the default message level for VSI C for OpenVMS systems.
  - LEVEL4 – Useful check/portable messages.
  - LEVEL5 – Not so useful check/portable messages.
  - LEVEL6 – Additional "noisy" messages.

  Be aware that there is a core of very important compiler messages that are enabled by default, regardless of what you specify with /WARNINGS or `#pragma message`. Referred to as message level 0, it includes all messages issued in header files, and comprises what is known as the nostandard group. All other message levels add additional messages to this core of enabled messages.

  You cannot modify level 0 (You cannot disable it, enable it, change its severity, or change its EMIT_ONCE characteristic). However, you can modify individual messages in level 0, provided such modification is allowed by the action. For example, you can disable a Warning or Informational in level 0, or you can change an error in level 0 to a Fatal, and so on. (See restrictions on modifying individual messages.)

  Enabling a level also enables all the messages in the levels lower than it. So enabling LEVEL3 messages also enables messages in LEVEL2 and LEVEL1.

  Disabling a level also disables all the messages in the levels higher than it. So disabling LEVEL4 messages also disables messages in LEVEL5 and LEVEL6.

- A comma-separated list of message identifiers, group names, and messages levels, freely mixed, enclosed in parentheses.

5.4.14.2. #pragma message *option2*

The parameter *option2* must be one of the following keywords:

- `save` — Saves the current state of which messages are enabled and disabled.
- `restore` — Restores the previous state of which messages are enabled and disabled.

The `save` and `restore` options are useful primarily within header files.

5.4.14.3. #pragma message (*quoted-string*)

This form of `#pragma message` is provided for compatibility with Microsoft's `#pragma message` directive.

The `#pragma message` (*quoted-string*) form of this directive emits the specified string as a compiler message. For example, when the compiler encounters the following line in the source file:

```
#pragma message ("hello")
```

It emits:

```
#pragma message ("hello")
................^
%CC-I-SIMPLEMESSAGE, hello
at line number 1 in file DISK1$:[SMITH]TEST.C;1
```

This form of the pragma is subject to macro replacement. For example, the following is allowed:

```
#pragma message ("Compiling file " __FILE__)
```

5.4.15. #pragma module Directive

When you compile source files to create an object file, the compiler assigns the first of the file names specified in the compilation unit to the name of the object file. The compiler adds the .OBJ file extension to the object file. Internally, the OpenVMS system (the debugger and the librarian) recognizes the object module by the file name; the compiler also gives the module a version number of 1. For example, given the object file EXAMPLE.OBJ, the debugger recognizes the EXAMPLE object module.

To change the system-recognized module name and version number, use the `#pragma module` directive. The `#pragma module` directive is specific to VSI C for OpenVMS systems and is not portable.

You can find the module name and the module version number listed in the compiler listing file and the linker load map.

The `#pragma module` directive is equivalent to the VAX C compatible `#module` directive. The `#pragma module` directive may be used when compiling in any mode. Use `#module` only when compiling with the /STANDARD=VAXC qualifier.

The `#pragma module` directive has the following formats:

```
#pragma module identifier identifier
#pragma module identifier string
```

The first parameter must be a valid VSI C identifier. It specifies the module name to be used by the linker. The second parameter specifies the optional identification that appears on listings and in the

object file. It must be either a valid VSI C identifier of 31 characters or less, or a character-string constant of 31 characters or less.

Only one `#pragma module` directive can be processed per compilation unit, and that directive must appear before any C language text. The `#pragma module` directive can follow other directives, such as `#define`, but it must precede any function definitions or external data definitions.

The parameters in a `#pragma module` directive are subject to text replacement and can, therefore, contain references to identifiers defined in previous `#define` directives. The replacement occurs before the parameters are processed.

5.4.16. #pragma names Directive

The `#pragma names` preprocessor directive provides the same kinds of control over the mapping of external identifiers' object-module symbols as does the /NAMES command-line qualifier, and it uses the same keywords. But as a pragma, the controls can be applied selectively to regions of declarations.

This pragma should only be used in header files and is intended for use by developers who supply libraries and/or header files to their customers.

The pragma has a `save`/`restore` stack that is also managed by `#pragma environment`, and so it is well-suited for use in header files. The effect of `#pragma environment header_defaults` is to set NAMES to `uppercase,truncated`, which is the compiler default.

The `#pragma names` directive has the following format:

```
#pragma names stack-option
#pragma names case-option[, length-option]
#pragma names length-option[, case-option]
```

Where *stack-option* is one of the following keywords:

- `save` - save the current names state
- `restore` - restore a saved names state

case-option is one of the following keywords:

- `uppercase` - uppercase external names
- `as_is` - do not change case

*length-option*is one of the following keywords:

- `truncated` - truncate at 31 characters
- `shortened` - shorten to 31 using CRC

An important use for this feature is to make it easier to use the command-line option /NAMES=AS_IS. Both the C99 standard and the C++ standard require that external names be treated as case-sensitive, and 3rd party libraries and Java native methods are starting to rely on case-sensitivity (C99 requires a minimum of 31 characters significant, while C++ requires all characters significant). Therefore, the use of /NAMES=AS_IS is expected to become more widespread.

The C run-time library is implemented with all symbols duplicated, spelled both in uppercase and lowercase, to allow C programs compiled with any of the /NAMES= settings to work. But traditional

practice on OpenVMS systems, combined with compiler defaults of /NAMES=UPPER, has resulted in nearly all existing object libraries and shared images to contain all uppercase names (both in references and in definitions), even though C source code using these libraries typically declares the names in lowercase or mixed case. Usually, the header files to access these libraries contain macro definitions to replace lowercase names by uppercase names to allow client programs to be compiled /NAMES=AS_IS. But macro definitions are problematic because every external name has to have a macro.

The new pragma allows header files to specify just once that the external names they declare are to be uppercased in the object module, regardless of the NAMES setting used in the rest of the compilation. The NAMES setting in effect at the first declaration of an external name is the one that takes effect; therefore, the setting specified in a header file is not overridden by a subsequent redeclaration in the user's program (which might specify a different NAMES setting). Note that the automatic Prologue/Epilogue header-file inclusion feature described in *Section 1.7.4, "Prologue/Epilogue Files"* (in connection with pointer_size pragmas) can also be used to specify the NAMES setting for all headers in a given directory or text library, without having to edit each header directly.

5.4.17. #pragma optimize Directive

The `#pragma optimize` preprocessor directive sets the optimization characteristics of function definitions that follow the directive. It allows optimization-control options that are normally set on the command line for the entire compilation to be specified in the source file for individual functions.

The `#pragma optimize` directive has the following format:

```
#pragma optimize settings
#pragma optimize save
#pragma optimize restore
#pragma optimize command_line
```

Where *settings* is any combination of the following:

- *level settings*

  These set the optimization level. Specify the level as follows:

  `level=`*n*

  Where *n* is an integer from 0 to 5.

- *unroll settings*

  These control loop unrolling. Specify as follows:

  `unroll=`*n*

  Where *n* is a nonnegative integer.

- *ansi-alias settings*

  These control ansi-alias assumptions. Specify one of the following:

  ```
  ansi_alias=on
  ansi_alias=off
  ```

- *intrinsic settings*

  These control recognition of intrinsics: Specify one of the following:

```
intrinsics=on
intrinsics=off
```

White space is optional between the setting clauses and before and after the "=" in each clause. The pragma is not subject to macro replacement.

For more information on the optimization settings, see *Table 1.16, "/OPTIMIZE Qualifier Options"* in the description of the /OPTIMIZE qualifier in *Section 1.3.4, "CC Command Qualifiers"*.

Example:

```
#pragma optimize level=5 unroll=6
```

Note

- If the level=0 clause is present, it must be the only clause present.
- The `#pragma optimize` directive must appear at file scope, outside any function body.
- If `#pragma optimize` does not specify a setting for one of the optimization states, that state remains unchanged.
- When a function definition is encountered, it is compiled using the optimization settings that are current at that point in the source.
- When a function is compiled under level=0, the compiler will not inline that function. In general, when functions are inlined, the inlined code is optimized using the optimization controls in effect at the call site instead of using the optimization controls specified for the function being inlined.
- When the OpenVMS command line specifies /NOOPT (or /OPTIMIZE=LEVEL=0), the `#pragma optimize` directive has no effect (except that its arguments are still validated).
- The `#pragma optimize` directive controls most, but not all, optimizations performed by the compiler. Therefore, there can be some differences between setting the optimization using the pragma compared with using the /OPTIMIZE command-line qualifier.

The `save` and `restore` options save and restore the current optimization state (level, unroll count, ansi-alias setting, and intrinsic setting).

The `command_line` option sets the optimization settings to what was specified on the command line.

5.4.18. #pragma pack Directive

The `#pragma pack` preprocessor directive specifies the byte boundary for packing members of C structures.

The `#pragma pack` directive has the following format:

```
#pragma pack n
#pragma pack ()
```

The *n* specifies the new alignment restriction in bytes:

1	align to byte

2	align to word
4	align to longword
8	align to quadword
16	align to octaword

A structure member is aligned to either the alignment specified by `#pragma pack` or the alignment determined by the size of the structure member, whichever is smaller. For example, a short variable in a structure gets byte-aligned if `#pragma pack 1` is specified, but word-aligned if `#pragma pack 2`, `4`, or `8` is specified.

When `#pragma pack` is specified without a value or with a value of 0, packing reverts to the /[NO]MEMBER_ALIGNMENT qualifier setting (either explicitly specified or by default) on the command line. Note that when specifying `#pragma pack` without a value, you must use parentheses: `#pragma pack ()`.

VSI C also supports the Microsoft Visual C++ enhanced syntax of this pragma:

```
#pragma pack ( { [ {push|pop} [ ,identifier ] [ ,n ] ] | [ n ] } )
```

With this enhanced syntax, you can save and restore packing alignment values across program components. This allows you to combine components into a single translation unit even if they specify different packing alignments:

- Every occurrence of `pragma pack` with a `push` argument stores the current packing alignment value on an internal compiler stack. If you provide a value for *n*, that value becomes the new packing value. If you specify an *identifier*, a name of your choosing, it is associated with the new packing value.
- Every occurrence of a `pragma pack` with a `pop` argument retrieves the value at the top of the stack and makes that value the new packing alignment. If an empty stack is popped, the alignment value defaults to the /[NO]MEMBER_ALIGNMENT command-line setting, and a warning is issued. If you specify a value for *n*, that value becomes the new packing value.

  If you specify an *identifier*, all values stored on the stack are removed from the stack until a matching *identifier* is found. The packing value associated with the *identifier* is also removed from the stack, and the packing value that was in effect just before the *identifier* was pushed becomes the new packing value. If no matching *identifier* is found, the packing value reverts to the command-line setting, and a warning is issued.

The enhanced syntax of `pragma pack` lets you write header files that ensure that packing values are the same before and after the header file is encountered. Consider the following example:

```
// File name: myinclude.h
//
#pragma pack( push, enter_myinclude )
// Your include-file code ...
#pragma pack( pop, enter_myinclude )
// End of myinclude.h
```

In this example, the current packing value is associated with the identifier `enter_myinclude` and pushed on entry to the header file. Your include code is processed. The `#pragma pack` at the end of the header file then removes all intervening packing values that might have occurred in the header file, as well as the packing value associated with `enter_myinclude`, thereby preserving the same packing value after the header file as before it.

The enhanced `pragma pack` syntax also lets you include header files that might set packing alignments different from the ones set in your code. Consider the following example:

```
#pragma pack( push, before_myinclude )
#include <myinclude.h>
#pragma pack( pop, before_myinclude )
```

In this example, your code is protected from any changes to the packing value that might occur in `<myinclude.h>` by saving the current packing alignment value, processing the include file (which may leave the packing alignment with an unknown setting), and restoring the original packing value.

5.4.19. #pragma pointer_size Directive

The `#pragma pointer_size` preprocessor directive can be used throughout a program to control whether pointers are 32-bit pointers or 64-bit pointers.

This directive has the same effect as the `#pragma required_pointer_size` directive, except that `#pragma pointer_size` is enabled only when the /POINTER_SIZE command-line qualifier is specified. If /POINTER_SIZE is omitted from the command line, `#pragma pointer_size` is ignored. (The `#pragma required_pointer_size` directive always takes effect, whether or not /POINTER_SIZE is specified.)

The `#pragma pointer_size` directive has the following format:

```
#pragma pointer_size keyword
```

The *keyword* is one of the following:

{`short`|32}	32-bit pointer
{`long`|64}	64-bit pointer
`system_default`	32-bit pointers on OpenVMS systems; 64-bit pointers on UNIX systems
`save`	Saves the current pointer size
`restore`	Restores the current pointer size to its last saved state

Notes

- The `#pragma pointer_size` and `#pragma required_pointer_size` directives only affect the meaning of the pointer-declarator (*) in declarations, casts, and the `sizeof` operator.
- The size of a pointer is the property of the type, and so it is bound in a `typedef` declaration, but not in a preprocessor macro definition.
- The size of a pointer produced by the `&` operator, or by an array name or function name in a context where it is converted to an explicit pointer, is 32 bits unless the `&` operator is applied to an object designated by a dereference of a pointer having a 64-bit pointer type.

5.4.20. #pragma required_pointer_size Directive

The `#pragma required_pointer_size` preprocessor directive is intended for use by developers of header files to control the size of pointers within a header file in those cases where the

pointers are architecturally required to be a particular size, and must not be altered by the user's use of pointer-size controls.

This directive has the same effect as the `#pragma pointer_size` directive, except that a `#pragma required_pointer_size` always takes effect, even if /POINTER_SIZE is omitted from the command line. (The `#pragma pointer_size` directive is ignored if /POINTER_SIZE is omitted.)

The `#pragma required_pointer_size` directive has the following format:

```
#pragma required_pointer_size keyword
```

The *keyword* is one of the following:

{`short`|32}	32-bit pointer
{`long`|64}	64-bit pointer
`system_default`	32-bit pointers on OpenVMS systems; 64-bit pointers on UNIX systems
`save`	Saves the current pointer size
`restore`	Restores the current pointer size to its last saved state

Notes

- The `#pragma pointer_size` and `#pragma required_pointer_size` directives only affect the meaning of the pointer-declarator (*) in declarations, casts, and the `sizeof` operator.
- The size of a pointer is the property of the type, and so it is bound in a `typedef` declaration, but not in a preprocessor macro definition.
- The size of a pointer produced by the `&` operator, or by an array name or function name in a context where it is converted to an explicit pointer, is 32 bits unless the `&` operator is applied to an object designated by a dereference of a pointer having a 64-bit pointer type.

5.4.21. #pragma [no]standard Directive

Use the `nostandard` and `standard` pragmas together to define regions of source code where portability diagnostics are not to be issued.

This pragma has the following format:

```
#pragma [no]standard
```

Use `#pragma nostandard` to suppress diagnostics about nonstandard extensions, regardless of the /STANDARD qualifier specified.

Use `#pragma standard` to direct the compiler to reinstate the setting of the /STANDARD qualifier that was in effect before the last `#pragma nostandard` was encountered. Every `#pragma standard` directive must be preceded by a corresponding `#pragma nostandard` directive.

The following example demonstrates the use of these pragmas:

```
#include <stdio.h>
```

```
#pragma nostandard
extern noshare FILE *stdin, *stdout, *stderr;
#pragma standard
```

In this example, `nostandard` prevents the NOSHAREEXT diagnostic from being issued against the `noshare` storage-class modifier, which is specific to VSI C for OpenVMS systems.

Note

This pragma does not change the current mode of the compiler or enable any extensions not already supported in that mode.

5.4.22. #pragma unroll Directive

Use the `#pragma unroll` preprocessor directive to unroll the `for` loop that follows it by the number of times specified in *unroll_factor*. The `#pragma unroll` directive must be followed by a `for` statement.

This pragma has the following format:

```
#pragma unroll (unroll_factor)
```

The *unroll_factor* is an integer constant in the range of 0 to 255. If a value of 0 is specified, the compiler ignores the directive and determines the number of times to unroll the loop in its normal way. A value of 1 prevents the loop from being unrolled. The directive applies only to the `for` loop that follows it, not to any subsequent for loops.

5.4.23. #pragma use_linkage Directive

After defining a special linkage using the `#pragma linkage` directive, described in *Section 5.4.11, "#pragma linkage Directive* (Alpha only)", use the `#pragma use_linkage` directive to associate the linkage with a function.

This pragma has the following format:

```
#pragma use_linkage linkage-name (id1, id2, ...)
```

The *linkage-name* is the name of a linkage previously defined by the `#pragma linkage` directive.

id1, *id2*, ... are the names of functions, or `typedef` names of function type, that you want associated with the specified linkage.

If you specify a `typedef` name of function type, then functions or pointers to functions declared using that type will have the specified linkage.

The `#pragma use_linkage` directive must appear in the source file before any use or definition of the specified routines. Otherwise, the results are unpredictable.

1.
```
#pragma linkage example_linkage = (parameters(r16, r17, r19),
 result(r16))
#pragma use_linkage example_linkage (sub)
int sub (int p1, int p2, short p3);

main()
{
```

```
    int result;

    result = sub (1, 2, 3);
}
```

This example defines a special linkage and associates it with a routine that takes three integer parameters and returns a single integer result in the same location where the first parameter was passed.

The `result (r16)` option indicates that the function result will be returned in R16 rather than the usual location (R0). The `parameters` option indicates that the three parameters passed to `sub` should be passed in R16, R17, and R19.

2.
```
#pragma linkage foo = (parameters(r1), result(r4))
#pragma use_linkage foo(f1,t)

int f1(int a);
typedef int t(int a);

t *f2;

#include <stdio.h>

main() {
    f2 = f1;
    b = (*f2)(1);
}
```

In this example, both the function `f1` and the function type `t` are given the linkage `foo`. The invocation through the function pointer `f2` will correctly invoke the function `f1` using the special linkage.

Chapter 6. Predefined Macros and Built-In Functions

This chapter describes the following topics:

- Predefined macros (*Section 6.1, "Predefined Macros"*)
- Built-in functions (*Section 6.2, "Built-In Functions"*)
  - For OpenVMS Alpha systems (Alpha only) (*Section 6.2.1, "Built-In Functions for OpenVMS Alpha Systems* (Alpha only)*"*)
  - For OpenVMS IA-64 Systems (I64 only) (*Section 6.2.2, "Built-In Functions for IA-64 systems* (I64 only)*"*)
  - For OpenVMS VAX systems (VAX only) (*Section 6.2.3, "Built-In Functions for OpenVMS VAX Systems* (VAX only)*"*)

Predefined macros and built-in functions are extensions to the C Standard and are specific to VSI C for OpenVMS systems. The macros assist in transporting code and performing simple tasks that are common to many programs. The built-in functions allow you to efficiently access processor instructions.

6.1. Predefined Macros

In addition to the standard-conforming, implementation-independent macros described in the *VSI C Reference Manual* [https://docs.vmssoftware.com/vsi-c-language-reference-manual/], VSI C for OpenVMS systems provides the predefined macros described in the following sections.

6.1.1. CC$gfloat (G_Floating Identification Macro)

This macro is provided for compatibility with VAX C. The `__G_FLOAT` predefined macro should be used instead. See *Section 6.1.4, "Floating-Point Macros "*.

6.1.2. System Identification Macros

Each implementation of the VSI C compiler automatically defines macros that can be used to identify the system on which the program is running. These macros can assist in writing code that executes conditionally, depending on the architecture or operating system on which the program is running.

Table 6.1, "Predefined System Identification Macros" lists the traditional (nonstandard) and new (standard) spellings of these predefined macro names for VSI C for OpenVMS Systems. Both spellings are defined for each macro unless strict ANSI C mode (/STANDARD=ANSI89) is in effect, in which case only the new spellings are defined.

Table 6.1. Predefined System Identification Macros

	Traditional Spelling	New Spelling
Operating system name:	`vms`	`__vms`
	`VMS`	`__VMS`
	`vms_version`	`__vms_version`

	Traditional Spelling	New Spelling
	VMS_VERSION	__VMS_VERSION
		__VMS_VER
		__DECC_VER
Architecture name:	vax (VAX only)	__vax (VAX only)
	VAX (VAX only)	__VAX (VAX only)
		__alpha *(Alpha only)*
		__ALPHA *(Alpha only)*
		__Alpha_AXP (Alpha only)
		__32BITS
		__ia64 (I64 only)
		__ia64__ (I64 only)
Product name:	vaxc	__vaxc
	VAXC	__VAXC
	vax11c	__vax11c
	VAX11C	__VAX11C
		__DECC
Standard C version of the compiler:		__STDC__ [1]
Compiler is a hosted implementation		__STDC_HOSTED__=1
ISOC94 version of the compiler		__STDC_VERSION__=199409L[2]
ISO/IEC 10646		__STDC_ISO_10646__=*yyyymmL*[3]
MIA version of the compiler:		__MIA [4]

[1] __STDC__ is defined only in strict or relaxed mode, and MIA mode.

[2] __STDC_VERSION__ is defined only when compiling with /STANDARD=ISOC94

[3] __STDC_ISO_10646__ evaluates to an integer constant of the form *yyyymmL* (for example, 199712L), intended to indicate that values of type wchar_t are the coded representations of the characters defined by ISO/IEC 10646, along with all amendments and technical corrigenda as of the specified year and month.

[4] __MIA is defined only in MIA mode.

Most of these macros are defined as 1 or 0, as appropriate to the processor and compilation qualifiers. Refer to the end of the compiler's source listing to see the names and values of all the macros that are defined prior to processing the first line of source code. The listing shows all macros predefined by the compiler, as well as those defined on the command line by the /DEFINE qualifier, but omits any that were undefined by the /UNDEFINE qualifier.

Note

Some users have tried defining the macro __ALPHA explicity with a /DEFINE qualifier or in a header file as a quick hack to deal with source-code conditionals that were written to assume that if __ALPHA is not defined, then the target must be a VAX. Doing this causes the CRTL headers and other OpenVMS

headers to take the wrong path for IA-64 systems. Never define any of the Alpha architecture predefined macros when using the compiler on IA-64 systems.

You can use these system identification macros to separate portable and nonportable code in any of your VSI C programs or to conditionally compile VSI C programs used on more than one operating system to take advantage of system-specific features. For example:

```
#ifdef    VMS
#include  rms           /* Include RMS definitions.  */
#endif
```

See the *VSI C Reference Manual* [https://docs.vmssoftware.com/vsi-c-language-reference-manual/] for more information about using the preprocessor conditional-compilation directives.

6.1.2.1. The __DECC_VER Macro

The `__DECC_VER` macro provides an integer encoding of the compiler version-identifier string that is suitable for use in a preprocessor `#if` expression, such that a larger number corresponds to a more recent version.

The format of the compiler version-identifier string is:

TMM.mm-eee

Where:

- *T* is the version type (letter).
- *MM* is the major version number.
- *mm* is the update (minor version number).
- *eee* is the edit suffix number.

The format of the integer encoding for `__DECC_VER` is:

vvuuteeee

Where:

- *vv* is the major version number.
- *uu* is the update (minor version number).
- *t* is the numerical encoding of the alphabetic version type from the version-identifier string.

  Table 6.2, "__DECC_VER Version-Type Encodings" lists the possible version types and their encodings:

Table 6.2. __DECC_VER Version-Type Encodings

Type	Numerical Encoding	Description
T	6	Field-test version
S	8	Customer special
V	9	Officially supported version

- *eeee* is the edit suffix number.

The following describes how the __DECC_VER integer value is calculated from the compiler version-identifier string:

1. The major version is multiplied by 10000000.
2. The minor version (the digits between the '.' and any edit suffix) is multiplied by 100000 and added to the suffix value (The suffix value has a range of 0-999).
3. If the character immediately preceding the first digit of the major version number is one of the ones listed in *Table 6.2, "__DECC_VER Version-Type Encodings"*, its numerical encoding is multiplied by 10000.
4. The preceding values are added together.

The following examples show how different compiler version-identifier strings map to __DECC_VER encodings:

```
ident            __DECC_VER
string           vvuuteeee

T5.2-003    ->   50260003
V6.0-001    ->   60090001
```

6.1.2.2. The __VMS_VER Macro

The __VMS_VER macro provides an integer encoding of the OpenVMS version-identifier string that is suitable for use in a preprocessor #if expression, such that a larger number corresponds to a more recent version.

The format of the OpenVMS version-identifier string is:

TMM.mm-epp

Where:

- *T* is the version type (letter).
- *MM* is the major version number.
- *mm* is the update (minor version number).
- *ee* is the edit number.
- *pp* is the patch letter.

The format of the integer encoding for __VMS_VER is:

vvuuepptt

Where:

- *vv* is the major version
- *uu* is the update (minor version)

- *e* is the edit number
- *pp* is the patch letter (A = 01, ..., Z = 26)
- *tt* is the alphabetic ordinal of the version type from the version-identifier string (E = 05, ..., V = 22)

  Note that there are no version-type letters A - D and W - Z.

The following describes how the __VMS_VER integer value is calculated from the OpenVMS version-identifier string:

1. The major version is multiplied by 10000000.
2. The minor version (the digits between the '.' and any edit/patch suffix) is multiplied by 100000 and added to the suffix value.

   The suffix value is the optional edit number multiplied by 10000, added to the optional patch letter's alphabetic ordinal multiplied by 100.
3. The preceding values are added together, along with the alphabetic ordinal of the version type.

The following examples show how different OpenVMS version-identifier strings map to __VMS_VER encodings:

```
ident           __VMS_VER
string          vvuuepptt

V6.1      ->    60100022
V6.1-1H   ->    60110822
E6.2      ->    60200005  ("IFT")
F6.2      ->    60200006  ("FT1")
G6.2      ->    60200007  ("FT2")
V6.2      ->    60200022
T6.2-1H   ->    60210820
V6.2-1I   ->    60210922
V5.5-1H1  ->    50510822 (extra trailing digit ignored)
```

6.1.3. Standards Conformance Macros

The VSI C RTL contains functions whose support and syntax conform to various industry standards or levels of product or operating system support.

Table 6.3, "Standards Macros – All platforms" lists macros that you can explicitly define (using the /DEFINE qualifier or the #define preprocessor directive) to control which VSI C RTL functions are declared in header files and to obtain standards conformance checking.

Table 6.3. Standards Macros – All platforms

Macro	Standard
_XOPEN_SOURCE_EXTENDED	XPG4-UNIX
_XOPEN_SOURCE	XPG4
_POSIX_C_SOURCE	POSIX
_ANSI_C_SOURCE	Standard C

Macro	Standard
_VMS_V6_SOURCE	OpenVMS Version 6 compatibility
_DECC_V4_SOURCE	DEC C Version 4.0 compatibility
_BSD44_CURSES	4.4BSD Curses
_VMS_CURSES	VAX C Curses
_SOCKADDR_LEN	4.4BSD sockets

These macros, with the exception of _POSIX_C_SOURCE, can be defined to 0 or 1.

The _POSIX_C_SOURCE macro can be defined to one of the following values:

0
1
2
199506

See the *VSI C Run-Time Library Reference Manual for OpenVMS Systems* [https://docs.vmssoftware.com/vsi-c-run-time-library-reference-manual-for-openvms-systems/] for more information about these feature-test macros.

6.1.4. Floating-Point Macros

VSI C for OpenVMS Systems automatically defines the following macros that pertain to the format of floating-point variables. They can be used to identify the format with which you are compiling your program.

- __D_FLOAT
- __G_FLOAT
- __IEEE_FLOAT
- _IEEE_FP
- __X_FLOAT

One of the first three macros listed is defined to have a value of 1 when the corresponding option of the /FLOAT qualifier is specified, or the appropriate /[NO]G_FLOAT qualifier is used. (The /G_FLOAT qualifier is kept only for compatibility with VAX C.) If the corresponding option was not specified, the associated macro is defined to have a value of 0.

The _IEEE_FP macro is defined in any IEEE floating-point mode except FAST.

On OpenVMS Alpha and IA-64 systems, the __X_FLOAT macro is defined to have a value of 1 when /L_DOUBLE_SIZE=128 (the default), and a value of 0 when /L_DOUBLE_SIZE=64.

These macros can assist in writing code that executes conditionally, depending on whether the program is running using D_floating, G_floating, or IEEE_floating precision.

For example, if you compiled using G_floating format, then __D_FLOAT and __IEEE_FLOAT are predefined to be 0, and __G_FLOAT is predefined as if the following were included before every compilation unit:

```
#define  __G_FLOAT  1
```

You can conditionally assign values to variables of type `double` without causing an error and without being certain of how much storage was allocated for the variable. For example, you may assign values to external variables as follows:

```
#ifdef __G_FLOAT
double x = 0.12e308;        /* Range to 10 to the 308th power */
#else
double x = 0.12e38;         /* Range to 10 to the 38th power  */
#endif
```

All predefined macro names, such as `__G_FLOAT`, are reserved by VSI.

You can remove the effect of predefined macro definitions by explicitly undefining the conflicting name. For more information about undefining macros, see the `#undefine` directive in the *VSI C Reference Manual* [https://docs.vmssoftware.com/vsi-c-language-reference-manual/]. For more information about the G_floating representation of the `double` data type, see *Chapter 4, "Data Storage and Representation"*.

6.1.5. Compiler-Mode Macros

The following predefined macros are defined if the corresponding compiler mode is selected:

- `__DECC_MODE_STRICT`
- `__DECC_MODE_RELAXED`
- `__DECC_MODE_VAXC`
- `__DECC_MODE_COMMON`
- `__DECC_MODE_MS`
- `__MS`

6.1.6. Pointer-Size Macro

The following predefined macro is defined if the /POINTER_SIZE command-line qualifier is specified:

`__INITIAL_POINTER_SIZE`

Specifying /POINTER_SIZE, /POINTER_SIZE=32, or /POINTER_SIZE=SHORT defines `__INITIAL_POINTER_SIZE` to 32.

Specifying /POINTER_SIZE=64, or /POINTER_SIZE=LONG defines `__INITIAL_POINTER_SIZE` to 64.

If /POINTER_SIZE is not specified, `__INITIAL_POINTER_SIZE` is defined to 0. This lets you use `#ifdef __INITIAL_POINTER_SIZE` to test whether or not the compiler supports 64-bit pointers, because compilers lacking pointer-size controls will not define this macro at all.

6.1.7. The __HIDE_FORBIDDEN_NAMES Macro

The C standard specifies exactly what identifiers in the normal name space are declared by the standard header files. A compiler is not free to declare additional identifiers in a header file unless the identifiers

follow defined rules (the identifier must begin with an underscore followed by an uppercase letter or another underscore).

When running the VSI C compiler for OpenVMS systems in strict ANSI C mode (/STANDARD=ANSI89), versions of the standard header files are included that hide many identifiers that do not follow the rules. The header file `<stdio.h>`, for example, hides the definition of the macro TRUE. The compiler accomplishes this by predefining the macro `__HIDE_FORBIDDEN_NAMES` in strict ANSI mode.

You can use the /UNDEFINE="__HIDE_FORBIDDEN_NAMES" command-line qualifier to prevent the compiler from predefining this macro and, thereby, including macro definitions of the forbidden names.

The header files are modified to only define additional VAX C names if `__HIDE_FORBIDDEN_NAMES` is undefined. For example, `<stdio.h>` might contain the following:

```
#ifndef __HIDE_FORBIDDEN_NAMES
#define TRUE 1
#endif
```

6.2. Built-In Functions

Sections *Section 6.2.1, "Built-In Functions for OpenVMS Alpha Systems* (Alpha only)*"*, *Section 6.2.2, "Built-In Functions for IA-64 systems* (I64 only)*"*, and *Section 6.2.3, "Built-In Functions for OpenVMS VAX Systems* (VAX only)*"* describe the VSI C built-in functions available in all compiler modes on OpenVMS Alpha, I64, and VAX systems.

These functions allow you to directly access hardware and machine instructions to perform operations that are cumbersome, slow, or impossible in other C compilers.

These functions are very efficient because they are built into the VSI C compiler. This means that a call to one of these functions does not result in a reference to a function in the VSI C Run-Time Library (C RTL) or to a function in your program. Instead, the compiler generates the machine instructions necessary to carry out the function directly at the call site. Because most of these built-in functions closely correspond to single VAX or Alpha machine instructions, the result is small, fast code.

Some of these built-in functions (such as those that operate on strings or bits) are of general interest. Others (such as the functions dealing with process context) are of interest if you are writing device drivers or other privileged software. Some of the functions discussed in the following sections are privileged and unavailable to user mode programs.

Be sure to include the `<builtins.h>` header file in your source program to access these built-in functions. VAX C required you to place the `#pragma builtins` preprocessor directive, rather than `#include <builtins.h>`, in your source file before using one or more built-in functions. VSI C supports `#pragma builtins` for compatibility with VAX C, but using `#include <builtins.h>` is recommended.

Note

VSI C implements `#pragma builtins` as if it were `#include <builtins.h>`; if you get an error from `#pragma builtins`, it is the same kind of error you would get if you specified `#include <builtins.h>`.

Also see *Section 5.4.2, "#pragma builtins Directive"*.

Some of the built-in functions have optional arguments or allow a particular argument to have one of many different types. To describe all valid combinations of arguments, the following built-in function descriptions list several different prototypes for the function. As long as a call to a built-in function matches one of the prototypes listed, the call is valid. Furthermore, any valid call to a built-in function behaves as if the corresponding prototype were in scope of the call. The compiler, therefore, performs the argument checking and conversions specified by that prototype.

The majority of the built-in functions are named after the processor instruction that they generate. The built-in functions provide direct and unencumbered access to those VAX instructions. Any inherent limitations to those instructions are limitations to the built-in functions as well. For instance, the MOVC3 instruction and the _MOVC3 built-in function can move at most 65,535 characters.

For more information on these built-in functions, see the corresponding machine instruction in the *VAX MACRO and Instruction Set Reference Manual*, *Alpha Architecture Handbook* or *Alpha Architecture Reference Manual*. In particular, refer to the structure of queue entries manipulated by the built-in queue functions.

6.2.1. Built-In Functions for OpenVMS Alpha Systems (Alpha only)

The following sections describe the VSI C built-in functions available on OpenVMS Alpha systems.

6.2.1.1. Translation Macros for VAX C Built-in Functions

On VSI C for OpenVMS Alpha Systems, the `<builtins.h>` header file contains macro definitions that translate some VAX C built-in functions to the equivalent VSI C for OpenVMS Alpha built-in functions. Consequently, the following VAX C built-in functions are effectively supported:

_BBCCI
_BBSSI
_INSQHI
_INSQTI
_INSQUE
_REMQHI
_REMQTI
_REMQUE
_PROBER
_PROBEW

For more detail on any of these functions, see `<builtins.h>` or the description of the corresponding native Alpha function in this chapter. For example, for a description of _INSQHI, see __PAL_INSQHIL.

6.2.1.2. In-line Assembly Code – ASMs

VSI C supports in-line assembly code, commonly referred to as ASMs on UNIX platforms.

Like built-in functions, ASMs are implemented with a function-call syntax. But unlike built-in functions, to use ASMs you must include the `<c_asm.h>` header file containing prototypes for the three types of ASMs, and the `#pragma intrinsic` preprocessor directive.

These functions have the following format:

```
__int64@@asm (const char *, …);  /* for integer operations, like MULQ */
float fasm    (const char *, …);  /* for single precision float
                                                    instructions, like MULS
 */
double dasm   (const char *, …);  /* for double precision float
                                                    instructions, like MULT
 */

#pragma intrinsic (asm, fasm, dasm)
```

const char *

The first argument to the `asm`, `fasm`, or `dasm` function contains the instruction(s) to be generated inline and the metalanguage that describes the interpretation of the arguments.

…

The source and destination arguments (if any) for the instruction being generated, and any other values used in the generated instructions.

These values are made available to the instructions through the normal argument passing conventions of the calling standard (the first integer argument is available in register R16).

The `#pragma intrinsic` directive in the `<c_asm.h>` header file is required when using ASMs. It notifies the compiler that:

- These functions are not user-defined functions.
- The special ASM processing should be applied to analyze at compile time the first argument and generate machine-code instructions as specified by the contents of the string.

The metalanguage for the argument references has the following form:

```
<metalanguage_sequence> : <register_alias>
                        | <register_number>
                        | <register_macro>
                        ;

<register_number>       : "$" number
                        ;

<register_macro>        : "%" <macro_sequence>
                        ;

<macro_sequence>        : number
                        | <register_name>
                        | "f" number | "F" number
                        | "r" number | "R" number
                        ;

<register_name> :     /* argument registers: R16-R21 */
                      "a0" | "a1" | "a2" | "a3" | "a4" | "a5"

                      /* return value: R0 or F0, depending on type */
                    | "v0"
```

```
                    /* scratch registers: R1, R22-R24, R28 */
                  | "t0" | "t1" | "t2" | "t3" | "t4"

                    /* save registers: R2-R15 */
                  | "s0" | "s1" | "s2" | "s3" | "s4"  | "s5"  | "s6"  |
"s7"
                  | "s8" | "s7" | "s8" | "s9" | "s10" | "s11" | "s12" |
"s13"

                    /* stack pointer: R30 */
                  | "sp" | "SP" | "$sp" | "$SP"

                  | "RA" | "ra"           /* return addr:        R26  */
                  | "PV" | "pv"           /* procedure value:    R27  */
                  | "AI" | "ai"           /* arg info:           R25  */
                  | "FP" | "fp"           /* frame pointer:      R29  */
                  | "RZ" | "rz" | "zero"  /* sink/source: R31 == zero */
```

Syntactically, the metalanguage can appear anywhere within an instruction sequence.

The literal string that contains instructions, operands, and metalanguage must follow the general form:

```
<string_contents>       :  <instruction_seq>
                        |  <string_contents> ";" <instruction_seq>
                        |  error
                        |  <string_contents> error
                        ;

<instruction_seq>       :  instruction_operand
                        |  directive
                        ;
```

An instruction_operand is generally recognized as an assembly language instruction separated by white space from a sequence of comma-separated operands.

You can code multiple instruction sequences into one literal string, separating them by semicolons.

Since the C language concatentates adjacent string literals into a single string, successive instructions can be written as separate strings, one per line (as is normally done in assembly language) as long as each instruction is terminated by a semicolon (as shown in the examples).

There are semantic and syntax rules associated with ASMs:

- The first argument to an ASM call is interpreted as the instructions to be assembled in the metalanguage, and must be fully understood by the compiler at compile time. Therefore, it must be a literal string (or a macro expanding to a literal string) and must not be a run-time value containing a string. Therefore, the following are not allowed: indirections, table lookups, structure dereferences, and so on.

- The remaining arguments are loaded into the argument registers like normal function arguments, except that the *second* argument to the ASM call is treated as the *first* argument for purposes of the calling standard.

  For example, in the following test, the six arguments are loaded into arg registers a0 through a5, and the result of each subexpression is stored in the value return register v0. Since v0 is the calling standard's return value register (R0 for an integer function), the result of the final MULQ is the value returned by the "call":

```
if (asm("mulq %a0, %a1,  %v0;"
        "mulq %a2, %v0,  %v0;"
        "mulq %a3, %v0,  %v0;"
        "mulq %a4, %v0,  %v0;"
        "mulq %a5, %v0,  %v0;", 1, 2, 3, 4, 5, 6) != 720){
  error_cnt++;
  printf ("Test failed\n");
     }
```

The following example does not work. There is no value loaded into the floating-point return register. Furthermore, it results in a compile-time warning stating that r2 is used before it is set, because the arguments are loaded into the arg registers and not into r2:

```
z =  fasm("mulq %r2, %a1, %r5", x=10, y=5);
```

The correct way of doing this is to specify an argument register number in place of r2. A correct version of the above would be:

```
z = fasm("mulq   %a0, %a1, %a1;"
         "stq    %a1, 0(%a2);"
         "ldt    %f0, 0(%a2);"
         "cvtqf  %f0, %f0;",  x=10, y=5, &temp);
```

Note that the memory location used for the transfer from integer to floating-point register is made available to the asm code by passing as an argument the address of a variable allocated in the C code for that purpose.

- A return register must be specified in the metalanguage for the result to appear in the expected place.
- For instructions that do not take any argument and do not have a return type, leave out the arguments. For example:

  ```
  asm("MB");
  ```

6.2.1.3. Absolute Value (__ABS)

The __ABS built-in is functionally equivalent to its counterpart, `abs`, in the standard header file `<stdlib.h>`.

Its format is also the same:

```
#include <stdlib.h>
int  __ABS (int x);
```

This built-in does, however, offer performance improvements because there is less call overhead associated with its use.

If you include `<stdlib.h>`, the built-in is automatically used for all occurrences of `abs`. To disable the built-in, use `#undef abs`.

6.2.1.4. Acquire and Release Longword Semaphore (__ACQUIRE_SEM_LONG, __RELEASE_SEM_LONG)

The __ACQUIRE_SEM_LONG and __RELEASE_SEM_LONG functions provide a counted semaphore capability where the positive value of a longword is interpreted as the number of resources available.

The __ACQUIRE_SEM_LONG function loops until the longword has a positive value and then decrements it within a load-locked/store-conditional sequence; it then issues a memory barrier. This function returns 1 if the resource count was successfully decremented within the specified number of retries, and 0 otherwise. With no explicit retry count, the function does not return until it succeeds.

The __RELEASE_SEM_LONG function issues a memory barrier and then does an __ATOMIC_INCREMENT_LONG on the longword.

The __ACQUIRE_SEM_LONG function has the following formats:

```
int  __ACQUIRE_SEM_LONG (volatile void *address);
int  __ACQUIRE_SEM_LONG_RETRY (volatile void *address, int retry);
```

The __RELEASE_SEM_LONG function has the following format:

```
int  __RELEASE_SEM_LONG (volatile void *address);
```

address

The longword-aligned address of the resource count.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the longword.

6.2.1.5. Add Aligned Word Interlocked (__ADAWI)

The __ADAWI function adds its source operand to the destination. This function is interlocked against similar operations by other processors or devices in the system.

This function has the following format:

```
int  __ADAWI (short src, volatile short *dest);
```

src

The value to be added to the destination.

dest

A pointer to the destination. The destination must be aligned on a word boundary. (You can achieve alignment using the `_align` or `__align` storage-class modifier.)

The __ADAWI function returns a simulated VAX processor status longword (PSL), the lower 4 bits of which are significant. These 4 bits are the condition codes and are defined as follows:

- Bit 3 is the negative condition code (N bit).

  In general, it is set by negative result instructions. The bit is cleared by positive result or zero instructions. For those instructions that affect the bit according to a stored result, the N bit reflects the actual result even if the sign of the result is algebraically incorrect as a result of overflow.

- Bit 2 is the zero condition code (Z bit).

  Typically it is set by instructions that store an exactly zero result and cleared if the result is not zero. Again, this reflects the actual result even if overflow occurs.

- Bit 1 is the overflow condition code (V bit).

  In general, it is set after arithmetic operations in which the magnitude of the algebraically correct result is too large to be represented in the available space, and cleared after operations whose result fits. Instructions in which overflow is impossible or meaningless either clear the bit or leave it unaffected. Note that all overflow conditions that set the V bit can also cause traps if the appropriate trap enable bits are set.

- Bit 0 is the carry condition code (C bit).

  Usually it is set after arithmetic operations in which a carry out of, or borrow into, the most significant bit occurred. The bit is cleared after arithmetic operations that had no carry or borrow, and is either cleared or unaffected by other instructions.

6.2.1.6. Add Atomic Longword (__ADD_ATOMIC_LONG)

The __ADD_ATOMIC_LONG function adds the specified expression to the aligned longword pointed to by the address parameter within a

load-locked/store-conditional

code sequence.

This function has the following format:

```
int  __ADD_ATOMIC_LONG (void *address, int expression, ...);
```

address

The address of the aligned longword.

expression

An integer expression.

...

An optional retry count of type `int`. If specified, the retry count indicates the number of times the operation is attempted (which will be at least once, even if the count argument is 0). If the operation cannot be performed successfully in the specified number of retries, a value of 0 is returned. If the operation is successful, a value of 1 is returned.

Note

If the optional retry count is omitted, this function loops back for a retry unconditionally on failure. In this case, the function can never return a failure value. It either returns a value of 1 upon successful completion, or hangs in an endless failure loop.

6.2.1.7. Add Atomic Quadword (__ADD_ATOMIC_QUAD)

The __ADD_ATOMIC_QUAD function adds the specified expression to the aligned quadword pointed to by the address parameter within a load-locked/store-conditional code sequence.

This function has the following format:

```
int  __ADD_ATOMIC_QUAD (void *address, int expression, ...);
```

address

The address of the aligned quadword.

expression

An integer expression.

...

An optional retry count of type `int`. If specified, the retry count indicates the number of times the operation is attempted (which will be at least once, even if the count argument is 0). If the operation cannot be performed successfully in the specified number of retries, a value of 0 is returned. If the operation is successful, a value of 1 is returned.

Note

If the optional retry count is omitted, this function loops back for a retry unconditionally on failure. In this case, the function can never return a failure value. It either returns a value of 1 upon successful completion, or hangs in an endless failure loop.

6.2.1.8. Allocate Bytes from Stack (__ALLOCA)

The __ALLOCA function allocates *n* bytes from the stack.

This function has the following format:

```
void  *__ALLOCA (unsigned int n);
```

n

The number of bytes to be allocated.

A pointer to the allocated memory is returned.

6.2.1.9. AND Atomic Longword (__AND_ATOMIC_LONG)

The __AND_ATOMIC_LONG function performs a bit-wise or arithmetic AND of the specified expression with the aligned longword pointed to by the address parameter within a load-locked/store-conditional code sequence.

This function has the following format:

```
int  __AND_ATOMIC_LONG (void *address, int expression, ...);
```

address

The longword-aligned address of the data segment.

expression

An integer expression.

...

An optional retry count of type `int`. If specified, the retry count indicates the number of times the operation is attempted (which will be at least once, even if the count argument is 0). If the operation

cannot be performed successfully in the specified number of retries, a value of 0 is returned. If the operation is successful, a value of 1 is returned.

Note

If the optional retry count is omitted, this function loops back for a retry unconditionally on failure. In this case, the function can never return a failure value. It either returns a value of 1 upon successful completion, or hangs in an endless failure loop.

6.2.1.10. AND Atomic Quadword (__AND_ATOMIC_QUAD)

The __AND_ATOMIC_QUAD function performs a bit-wise or arithmetic AND of the specified expression with the aligned quadword pointed to by the address parameter within a load-locked/store-conditional code sequence.

This function has the following format:

```
int  __AND_ATOMIC_QUAD (void *address, int expression, ...);
```

address

The address of the aligned quadword.

expression

An integer expression.

...

An optional retry count of type `int`. If specified, the retry count indicates the number of times the operation is attempted (which will be at least once, even if the count argument is 0). If the operation cannot be performed successfully in the specified number of retries, a value of 0 is returned. If the operation is successful, a value of 1 is returned.

Note

If the optional retry count is omitted, this function loops back for a retry unconditionally on failure. In this case, the function can never return a failure value. It either returns a value of 1 upon successful completion, or hangs in an endless failure loop.

6.2.1.11. Atomic Add Longword (__ATOMIC_ADD_LONG)

The __ATOMIC_ADD_LONG function adds the specified expression to the aligned longword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the longword before the addition was performed.

This function has the following formats:

```
int  __ATOMIC_ADD_LONG (volatile void *address, int expression);
int  __ATOMIC_ADD_LONG_RETRY (volatile void *address, int expression,
int retry, int *status);
```

address

The longword-aligned address of the data segment.

expression

An integer expression.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the longword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

Note

The non-RETRY form of this function loops back for a retry unconditionally on failure. This means this function can hang in an endless failure loop.

6.2.1.12. Atomic Add Quadword (__ATOMIC_ADD_QUAD)

The __ATOMIC_ADD_QUAD function adds the specified expression to the aligned quadword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the quadword before the addition was performed.

This function has the following formats:

```
__int64  __ATOMIC_ADD_QUAD (volatile void *address, __int64 expression);
__int64  __ATOMIC_ADD_QUAD_RETRY (volatile void *address,
__int64 expression, int retry, int *status);
```

address

The quadword-aligned address of the data segment.

expression

An integer expression.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the quadword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

Note

The non-RETRY form of this function loops back for a retry unconditionally on failure. This means this function can hang in an endless failure loop.

6.2.1.13. Atomic AND Longword (__ATOMIC_AND_LONG)

The __ATOMIC_AND_LONG function performs a bit-wise or arithmetic AND of the specified expression with the aligned longword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the longword before the operation was performed.

This function has the following formats:

```
int  __ATOMIC_AND_LONG (volatile void *address, int expression);
int  __ATOMIC_AND_LONG_RETRY (volatile void *address, int expression,
int retry, int *status);
```

address

The longword-aligned address of the data segment.

expression

An integer expression.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the longword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

Note

The non-RETRY form of this function loops back for a retry unconditionally on failure. This means this function can hang in an endless failure loop.

6.2.1.14. Atomic AND Quadword (__ATOMIC_AND_QUAD)

The __ATOMIC_AND_QUAD function performs a bit-wise or arithmetic AND of the specified expression with the aligned quadword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the quadword before the operation was performed.

This function has the following formats:

```
__int64  __ATOMIC_AND_QUAD (volatile void *address, __int64 expression);
__int64  __ATOMIC_AND_QUAD_RETRY (volatile void *address,
__int64 expression, int retry, int *status);
```

address

The quadword-aligned address of the data segment.

expression

An integer expression.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the quadword.

`status`

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

Note

The non-RETRY form of this function loops back for a retry unconditionally on failure. This means this function can hang in an endless failure loop.

6.2.1.15. Atomic OR Longword (__ATOMIC_OR_LONG)

The __ATOMIC_OR_LONG function performs a bit-wise or arithmetic OR of the specified expression with the aligned longword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the longword before the operation was performed.

This function has the following formats:

```
int  __ATOMIC_OR_LONG (volatile void *address, int expression);
int  __ATOMIC_OR_LONG_RETRY (volatile void *address, int expression,
int retry, int *status);
```

`address`

The longword-aligned address of the data segment.

`expression`

An integer expression.

`retry`

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the longword.

`status`

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

Note

The non-RETRY form of this function loops back for a retry unconditionally on failure. This means this function can hang in an endless failure loop.

6.2.1.16. Atomic OR Quadword (__ATOMIC_OR_QUAD)

The __ATOMIC_OR_QUAD function performs a bit-wise or arithmetic OR of the specified expression with the aligned quadword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the quadword before the operation was performed.

This function has the following formats:

```
__int64  __ATOMIC_OR_QUAD (volatile void *address, __int64 expression);
__int64  __ATOMIC_OR_QUAD_RETRY (volatile void *address,
__int64 expression, int retry, int *status);
```

address

The quadword-aligned address of the data segment.

expression

An integer expression.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the quadword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

Note

The non-RETRY form of this function loops back for a retry unconditionally on failure. This means this function can hang in an endless failure loop.

6.2.1.17. Atomic Increment Longword (__ATOMIC_INCREMENT_LONG)

The __ATOMIC_INCREMENT_LONG function increments by 1 the aligned longword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the longword before the operation was performed.

This function has the following formats:

```
int  __ATOMIC_INCREMENT_LONG (volatile void *address);
int  __ATOMIC_INCREMENT_LONG_RETRY (volatile void *address, int retry, int
*status);
```

address

The longword-aligned address of the data segment.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the quadword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

6.2.1.18. Atomic Increment Quadword (__ATOMIC_INCREMENT_QUAD)

The __ATOMIC_INCREMENT_QUAD function increments by 1 the aligned quadword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the quadword before the operation was performed.

This function has the following formats:

```
__int64  __ATOMIC_INCREMENT_QUAD (volatile void *address);
__int64  __ATOMIC_INCREMENT_QUAD (volatile void *address, int retry, int
*status);
```

address

The quadword-aligned address of the data segment.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the quadword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

6.2.1.19. Atomic Decrement Longword (__ATOMIC_DECREMENT_LONG)

The __ATOMIC_DECREMENT_LONG function decrements by 1 the aligned longword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the longword before the operation was performed.

This function has the following formats:

```
int  __ATOMIC_DECREMENT_LONG (volatile void *address);
int  __ATOMIC_DECREMENT_LONG (volatile void *address, int retry, int
*status);
```

address

The longword-aligned address of the data segment.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the quadword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

6.2.1.20. Atomic Decrement Quadword (__ATOMIC_DECREMENT_QUAD)

The __ATOMIC_DECREMENT_QUAD function decrements by 1 the aligned quadword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the quadword before the operation was performed.

This function has the following formats:

```
__int64  __ATOMIC_DECREMENT_QUAD (volatile void *address);
__int64  __ATOMIC_DECREMENT_QUAD (volatile void *address, int retry, int
*status);
```

address

The quadword-aligned address of the data segment.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the quadword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

6.2.1.21. Atomic Exchange Longword (__ATOMIC_EXCH_LONG)

The __ATOMIC_EXCH_LONG function stores the value of the specified expression into the aligned longword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the longword before the operation was performed.

This function has the following formats:

```
int  __ATOMIC_EXCH_LONG        (volatile void *address, int expression);
int  __ATOMIC_EXCH_LONG_RETRY  (volatile void *address, int expression,
int retry, int *status);
```

address

The longword-aligned address of the data segment.

expression

An integer expression.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the longword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

Note

The non-RETRY form of this function loops back for a retry unconditionally on failure. This means this function can hang in an endless failure loop.

6.2.1.22. Atomic Exchange Quadword (__ATOMIC_EXCH_QUAD)

The __ATOMIC_EXCH_QUAD function stores the value of the specified expression into the aligned quadword pointed to by the *address* parameter within a load-locked/store-conditional code sequence and returns the value of the quadword before the operation was performed.

This function has the following formats:

```
__int64  __ATOMIC_EXCH_QUAD (volatile void *address,
__int64 expression);
__int64  __ATOMIC_EXCH_QUAD_RETRY (volatile void *address,
__int64 expression, int retry, int *status);
```

address

The quadword-aligned address of the data segment.

expression

An integer expression.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the quadword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

Note

The non-RETRY form of this function loops back for a retry unconditionally on failure. This means this function can hang in an endless failure loop.

6.2.1.23. Compare Store Longword (__CMP_STORE_LONG)

The __CMP_STORE_LONG function has the following format:

```
int  __CMP_STORE_LONG (volatile void *source, int old_value,
int new_value, volatile void *dest);
```

This function performs a conditional atomic compare and update operation involving one or two longwords in the same lock region. The value pointed to by *source* is compared with the longword

old_value. If they are equal, the longword *new_value* is conditionally stored into the value pointed to by *dest*.

The store will not complete if the compare yields unequal values or if there is an intervening store to the lock region involved. To be in the same lock region, *source* and *dest* must point to aligned longwords in the same naturally aligned 16-byte region.

The function returns 0 if the store does not complete, and returns 1 if the store does complete.

6.2.1.24. Compare Store Quadword (__CMP_STORE_QUAD)

The __CMP_STORE_QUAD function has the following format:

```
int  __CMP_STORE_QUAD (volatile void *source, int64 old_value,
int64 new_value, volatile void *dest);
```

This function performs a conditional atomic compare and update operation involving one or two quadwords in the same lock region. The value pointed to by *source* is compared with the quadword *old_value*. If they are equal, the quadword *new_value* is conditionally stored into the value pointed to by *dest*.

The store will not complete if the compare yields unequal values or if there is an intervening store to the lock region involved. To be in the same lock region, *source* and *dest* must point to aligned quadwords in the same naturally aligned 16-byte region.

The function returns 0 if the store does not complete, and returns 1 if the store does complete.

6.2.1.25. Convert G_Floating to F_Floating Chopped (__CVTGF_C)

The __CVTGF_C function converts a double-precision, VAX G_floating-point number to a single-precision, VAX F_floating-point number. This conversion chops to single-precision; then the 8-bit exponent range is checked for overflow or underflow.

This function has the following format:

```
float  __CVTGF_C (double operand);
```

operand

A double-precision, VAX floating-point number.

6.2.1.26. Convert G_Floating to Quadword (__CVTGQ)

The __CVTGQ function rounds a double-precision, VAX floating-point number to a 64-bit integer value and returns the result.

This function has the following format:

```
int64  __CVTGQ (double operand);
```

operand

A double-precision, VAX floating-point number.

6.2.1.27. Convert IEEE T_Floating to IEEE S_Floating Chopped (__CVTTS_C)

The __CVTTS_C function converts a double-precision, IEEE T_floating-point number to a single-precision, IEEE S_floating-point number. This conversion chops to single-precision; then the 8-bit exponent range is checked for overflow or underflow.

This function has the following format:

```
float  __CVTTS_C (double operand);
```

operand

A double-precision, IEEE floating-point number.

6.2.1.28. Convert IEEE T_Floating to Quadword (__CVTTQ)

The __CVTTQ function rounds a double-precision, IEEE T_floating-point number to a 64-bit integer value and returns the result.

This function has the following format:

```
int64  __CVTTQ (double operand);
```

operand

A double-precision, IEEE T_floating-point number.

6.2.1.29. Convert X_Floating to Quadword (__CVTXQ)

The __CVTXQ function converts an X_floating-point number to a 64-bit integer value and returns the result.

This function has the following format:

```
int64  __CVTXQ (long double operand);
```

operand

An X_floating-point number.

6.2.1.30. Convert X_Floating to IEEE T_Floating Chopped (__CVTXT_C)

The __CVTXT_C function converts an X_floating-point number to an IEEE

T_floating-point

number and returns the result.

This function has the following format:

```
double  __CVTXT_C (long double operand);
```

operand

An X_floating-point number.

6.2.1.31. Copy Sign Built-in Functions

Built-in functions are provided to copy selected portions of single- and double-precision, floating-point numbers.

These built-in functions have the following format:

```
float   __CPYSF (float operand1, float operand2);
double  __CPYS (double operand1, double operand2);

float   __CPYSNF (float operand1, float operand2);
double  __CPYSN (double operand1, double operand2);

float   __CPYSEF (float operand1, float operand2);
double  __CPYSE (double operand1, double operand2);
```

The copy sign built-ins (__CPYSF and __CPYS) fetch the sign bit in *operand1*, concatenate it with the exponent and fraction bits from *operand2*, and return the result.

The copy sign negate built-ins (__CPYSNF and __CPYSN) fetch the sign bit in *operand1*, complement it, concatenate it with the exponent and fraction bits from *operand2*, and return the result.

The copy sign exponent built-ins (__CPYSEF and __CPYSE) fetch the sign and exponent bits from *operand1*, concatenate them with the fraction bits from *operand2*, and return the result.

6.2.1.32. Cosine (__COS)

The __COS built-in function is functionally equivalent to its counterpart, `cos`, in the standard header file `<math.h>`.

Its format is also the same:

```
#include <math.h>
double  __COS (double x);
```

x

A radian value.

This built-in offers performance improvements because there is less call overhead associated with its use.

If you include `<math.h>`, the built-in is automatically used for all occurrences of `cos`. To disable the built-in, use `#undef cos`.

6.2.1.33. Double-Precision, Floating-Point Arithmetic Built-in Functions

The following built-in functions provide double-precision, floating-point chopped arithmetic:

__ADDG_C	__ADDT_C	__SUBG_C	__SUBT_C
__MULG_C	__MULT_C	__DIVG_C	__DIVT_C

They have the following format:

```
double  __op{G,T}_C (double operand1, double operand2);
```

Where *op* is one of ADD, SUB, MUL, DIV, and {G,T} represents VAX or IEEE floating-point arithmetic, respectively.

The result of the arithmetic operation is returned.

6.2.1.34. Floating-Point Absolute Value (__FABS)

The __FABS built-in function is functionally equivalent to its counterpart, `fabs`, in the standard header file `<math.h>`.

Its format is also the same:

```
#include <math.h>
double __FABS (double x);
```

x

A floating-point number.

This built-in offers performance improvements because there is no call overhead associated with its use.

If you include `<math.h>`, the built-in is automatically used for all occurrences of `fab`. To disable the built-in, use `#undef fab`.

6.2.1.35. _leadz

The _leadz built-in function returns the number of leading zeroes (starting at the most significant bit position) in its argument. For example, _leadz(1) returns 63, and _leadz(0) returns 64.

This function has the following format:

```
int64 _leadz (unsigned int64);
```

6.2.1.36. Long Double-Precision, Floating-Point Arithmetic Built-in Functions

The following built-in functions provide long double-precision, floating-point chopped arithmetic:

__ADDX_C	__SUBX_C
__MULX_C	__DIVX_C

They have the following format:

```
long double  __opX_C (long double operand1, long double operand2);
```

Where *op* is one of ADD, SUB, MUL, DIV.

The result of the arithmetic operation is returned.

6.2.1.37. Longword Absolute Value (__LABS)

The __LABS built-in is functionally equivalent to its counterpart, `labs`, in the standard header file `<stdlib.h>`.

Its format is also the same:

```
#include <stdlib.h>
long int  __LABS (long int x);
```

x

An integer.

This built-in offers performance improvements because there is less call overhead associated with its use.

If you include `<stdlib.h>`, the built-in is automatically used for all occurrences of `labs`. To disable the built-in, use `#undef labs`.

6.2.1.38. Lock and Unlock Longword (__LOCK_LONG, __UNLOCK_LONG)

The __LOCK_LONG and __UNLOCK_LONG functions provide a binary spinlock capability based on the low-order bit of a longword.

The __LOCK_LONG function executes in a loop waiting for the bit to be cleared and then sets it within a load-locked/store-conditional sequence; it then issues a memory barrier. The __UNLOCK_LONG function issues a memory barrier and then zeroes the longword.

The __LOCK_LONG_RETRY function returns 1 if the lock was acquired in the specified number of retries and 0 if the lock was not acquired.

The __LOCK_LONG function has the following formats:

```
int  __LOCK_LONG (volatile void *address);
int  __LOCK_LONG_RETRY (volatile void *address, int retry);
```

The __UNLOCK_LONG function has the following format:

```
int  __UNLOCK_LONG (volatile void *address);
```

address

The quadword-aligned address of the longword used for the lock.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the longword.

6.2.1.39. Memory Barrier (__MB)

The __MB function directs the compiler to generate a memory barrier instruction.

This function has the following format:

```
void  __MB (void);
```

6.2.1.40. Memory Copy and Set Functions (__MEMCPY, __MEMMOVE, __MEMSET)

The __MEMCPY, __MEMMOVE, and __MEMSET built-ins are functionally equivalent to their run-time routine counterparts in the standard header file `<string.h>`.

Their format is also the same:

```
#include <string.h>
void  *__MEMCPY  (void *s1, const void *s2, size_t size);
void  *__MEMMOVE (void *s1, const void *s2, size_t size);
void  *__MEMSET  (void *s, int value, size_t size);
```

These built-ins offer performance improvements because there is less call overhead associated with their use.

If you include `<string.h>`, the built-ins are automatically used for all occurrences of `memcpy`, `memmove`, and `memset`. To disable the built-ins, use `#undef memcpy`, `#undef memmove`, and `#undef memset`.

6.2.1.41. OR Atomic Longword (__OR_ATOMIC_LONG)

The __OR_ATOMIC_LONG function performs a bit-wise or arithmetic OR of the specified expression with the aligned longword pointed to by the address parameter within a load-locked/store-conditional code sequence.

This function has the following format:

```
int  __OR_ATOMIC_LONG (void *address, int expression, ...);
```

address

The address of the aligned longword.

expression

An integer expression.

...

An optional retry count of type `int`. If specified, the retry count indicates the number of times the operation is attempted (which will be at least once, even if the count argument is 0). If the operation cannot be performed successfully in the specified number of retries, a value of 0 is returned. If the operation is successful, a value of 1 is returned.

Note

If the optional retry count is omitted, this function loops back for a retry unconditionally on failure. In this case, the function can never return a failure value. It can either return a value of 1 upon successful completion, or it can hang in an endless failure loop.

6.2.1.42. OR Atomic Quadword (__OR_ATOMIC_QUAD)

The __OR_ATOMIC_QUAD function performs a bit-wise or arithmetic OR of the specified expression with the aligned quadword pointed to by the address parameter within a load-locked/store-conditional code sequence.

This function has the following format:

```
int  __OR_ATOMIC_QUAD (void *address, int expression, ...);
```

address

The address of the aligned quadword.

expression

An integer expression.

...

An optional retry count of type `int`. If specified, the retry count indicates the number of times the operation is attempted (which will be at least once, even if the count argument is 0). If the operation cannot be performed successfully in the specified number of retries, a value of 0 is returned. If the operation is successful, a value of 1 is returned.

Note

If the optional retry count is omitted, this function loops back for a retry unconditionally on failure. In this case, the function can never return a failure value. It can either return a value of 1 upon successful completion, or it can hang in an endless failure loop.

6.2.1.43. Privileged Architecture Library Code Instructions

The following sections describe the Privileged Architecture Library Code (PALcode) instructions that are available as built-in functions.

6.2.1.44. __PAL_BPT

This function is provided for program debugging. It switches the processor to kernel mode and pushes registers R2 through R7, the updated PC, and PS onto the kernel stack. It then dispatches to the address in the breakpoint vector, which is stored in a control block.

This function has the following format:

```
void  __PAL_BPT (void);
```

6.2.1.45. __PAL_BUGCHK

This function is provided for error reporting. It switches the processor to kernel mode and pushes registers R2 through R7, the updated PC, and PS onto the kernel stack. It then dispatches to the address in the bugcheck vector, which is stored in a control block.

This function has the following format:

```
void  __PAL_BUGCHK (unsigned __int64 code);
```

6.2.1.46. __PAL_CFLUSH

This function flushes at least the entire physical page specified by the page frame number *value* from any data caches associated with the current processor. After a CFLUSH is done, the first subsequent load on the same processor to an arbitrary address in the target page is fetched from physical memory.

This function has the following format:

```
void  __PAL_CFLUSH (int value);
```

value

A page frame number.

6.2.1.47. __PAL_CHME

This function allows a process to change its mode to Executive in a controlled manner. The change in mode also results in a change of stack pointers: the old pointer is saved and the new pointer is loaded. Registers R2 through R7, PS, and PC are pushed onto the selected stack. The saved PC addresses the instruction following the CHME instruction.

This function has the following format:

```
void  __PAL_CHME (void);
```

6.2.1.48. __PAL_CHMK

This function allows a process to change its mode to kernel in a controlled manner. The change in mode also results in a change of stack pointers: the old pointer is saved and the new pointer is loaded. Registers R2 through R7, PS, and PC are pushed onto the kernel stack. The saved PC addresses the instruction following the CHMK instruction.

This function has the following format:

```
void  __PAL_CHMK (void);
```

6.2.1.49. __PAL_CHMS

This function allows a process to change its mode to Supervisor in a controlled manner. The change in mode also results in a change of stack pointers: the old pointer is saved and the new pointer is loaded. Registers R2 through R7, PS, and PC are pushed onto the selected stack. The saved PC addresses the instruction following the CHMS instruction.

This function has the following format:

```
void  __PAL_CHMS (void);
```

6.2.1.50. __PAL_CHMU

This function allows a process to call a routine using the change mode mechanism. Registers R2 through R7, PS, and PC are pushed onto the current stack. The saved PC addresses the instruction following the CHMU instruction.

This function has the following format:

```
void  __PAL_CHMU (void);
```

6.2.1.51. __PAL_DRAINA

This function stalls instruction issuing until all prior instructions are guaranteed to complete without incurring aborts.

This function has the following format:

```
void  __PAL_DRAINA (void);
```

6.2.1.52. __PAL_GENTRAP

This function is used for reporting run-time software conditions.

This function has the following format:

```
void  __PAL_GENTRAP (uint64 encoded_software_trap);
```

encoded_software_trap

The particular software condition that has occurred.

6.2.1.53. __PAL_HALT

This function halts the processor when executed by a process running in kernel mode. This is a privileged function.

This function has the following format:

```
void  __PAL_HALT (void);
```

6.2.1.54. __PAL_INSQHIL

This function inserts an entry at the front of a longword queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system. This function must have write access to header and queue entries. The pointers to *head* and *new_entry* must not be equal.

This function has the following format:

```
int __PAL_INSQHIL (void *head, void *new_entry); /* At head, interlocked */
```

head

A pointer to the queue header. The header must be aligned on a quadword boundary.

new_entry

A pointer to the new entry to be inserted. The entry must be aligned on a longword boundary.

There are three possible return values:

- –1 if the entry was not inserted because the secondary interlock failed
- 0 if the entry was inserted but it was not the only entry in the list
- 1 if the entry was inserted and it was the only entry in the list

6.2.1.55. __PAL_INSQHILR

This function inserts an entry into the front of a longword queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system. This function must have write access to the header and queue entries. The pointers to *head* and *new_entry* must not be equal. All parts of the queue must be memory resident.

This function has the following format:

```
int __PAL_INSQHILR (void *head, void *new_entry);
/* At head, interlocked resident */
```

head

A pointer to the queue header. The header must be aligned on a quadword boundary.

new_entry

A pointer to the new entry to be inserted. The entry must be aligned on a quadword boundary.

There are three possible return values:

- –1 if the entry was not inserted because the secondary interlock failed
- 0 if the entry was inserted but it was not the only entry in the list
- 1 if the entry was inserted and it was the only entry in the list

6.2.1.56. __PAL_INSQHIQ

This function inserts an entry at the front of a quadword queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system. This function must have write access to header and queue entries. The pointers to *head* and *new_entry* must not be equal.

This function has the following format:

```
int __PAL_INSQHIQ (void *head, void *new_entry); /* At head, interlocked */
```

head

A pointer to the queue header. The header must be aligned on an octaword boundary.

new_entry

A pointer to the new entry to be inserted. The entry must be aligned on an octaword boundary.

There are three possible return values:

- –1 if the entry was not inserted because the secondary interlock failed
- 0 if the entry was inserted but it was not the only entry in the list
- 1 if the entry was inserted and it was the only entry in the list

6.2.1.57. __PAL_INSQHIQR

This function inserts an entry into the front of a quadword queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system. This function must have write access to the header and queue entries. The pointers to *head* and *new_entry* must not be equal. All parts of the queue must be memory resident.

This function has the following format:

```
int  __PAL_INSQHIQR (void *head, void *new_entry);
/* At head, interlocked resident */
```

head

A pointer to the queue header. The header must be aligned on an octaword boundary.

new_entry

A pointer to the new entry to be inserted. The entry must be aligned on an octaword boundary.

There are three possible return values:

- –1 if the entry was not inserted because the secondary interlock failed
- 0 if the entry was inserted but it was not the only entry in the list
- 1 if the entry was inserted and it was the only entry in the list

6.2.1.58. __PAL_INSQTIL

This function inserts an entry at the end of a longword queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system. This function must have write access to header and queue entries. The pointers to *head* and *new_entry* must not be equal.

This function has the following format:

```
int  __PAL_INSQTIL (void *head, void *new_entry);
/* At tail, interlocked */
```

head

A pointer to the queue header. The header must be aligned on a quadword boundary.

new_entry

A pointer to the new entry to be inserted. The entry must be aligned on a quadword boundary.

There are three possible return values:

- –1 if the entry was not inserted because the secondary interlock failed
- 0 if the entry was inserted but it was not the only entry in the list
- 1 if the entry was inserted and it was the only entry in the list

6.2.1.59. __PAL_INSQTILR

This function inserts an entry at the end of a longword queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system. This function must have write access to the header and queue entries. The pointers to *head* and *new_entry* must not be equal. All parts of the queue must be memory resident.

This function has the following format:

```
int  __PAL_INSQTILR (void *head, void *new_entry);
/* At tail, interlocked resident */
```

head

A pointer to the queue header. The header must be aligned on a quadword boundary.

new_entry

A pointer to the new entry to be inserted. The entry must be aligned on a quadword boundary.

There are three possible return values:

- –1 if the entry was not inserted because the secondary interlock failed

- 0 if the entry was inserted but it was not the only entry in the list
- 1 if the entry was inserted and it was the only entry in the list

6.2.1.60. __PAL_INSQTIQ

This function inserts an entry at the end of a quadword queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system. This function must have write access to header and queue entries. The pointers to *head* and *new_entry* must not be equal.

This function has the following format:

```
int  __PAL_INSQTIQ (void *head, void *new_entry);
/* At tail, interlocked */
```

head

A pointer to the queue header. The header must be aligned on an octaword boundary.

new_entry

A pointer to the new entry to be inserted. The entry must be aligned on an octaword boundary.

There are three possible return values:

- –1 if the entry was not inserted because the secondary interlock failed
- 0 if the entry was inserted but it was not the only entry in the list
- 1 if the entry was inserted and it was the only entry in the list

6.2.1.61. __PAL_INSQTIQR

This function inserts an entry at the end of a quadword queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system. This function must have write access to the header and queue entries. The pointers to *head* and *new_entry* must not be equal. All parts of the queue must be memory resident.

This function has the following format:

```
int  __PAL_INSQTIQR (void *head, void *new_entry);
/* At tail, interlocked resident */
```

head

A pointer to the queue header. The header must be aligned on an octaword boundary.

new_entry

A pointer to the new entry to be inserted. The entry must be aligned on an octaword boundary.

There are three possible return values:

- –1 if the entry was not inserted because the secondary interlock failed
- 0 if the entry was inserted but it was not the only entry in the list
- 1 if the entry was inserted and it was the only entry in the list

6.2.1.62. __PAL_INSQUEL

This function inserts a new entry after an existing entry into a longword queue. This function must have write access to header and queue entries.

This function has the following format:

```
int  __PAL_INSQUEL (void *predecessor, void *new_entry);
```

predecessor

A pointer to an existing entry in the queue.

new_entry

A pointer to the new entry to be inserted.

There are two possible return values:

- 0 if the entry was not the only entry in the queue
- 1 if the entry was the only entry in the queue

6.2.1.63. __PAL_INSQUEL_D

This function inserts a new entry after an existing entry into a longword queue deferred. This function must have write access to header and queue entries.

This function has the following format:

```
int __PAL_INSQUEL_D (void **predecessor, void *new_entry); /* Deferred */
```

predecessor

A pointer to a pointer to the predecessor entry.

new_entry

A pointer to the new entry to be inserted.

There are two possible return values:

- 0 if the entry was not the only entry in the queue
- 1 if the entry was the only entry in the queue

6.2.1.64. __PAL_INSQUEQ

This function inserts a new entry after an existing entry into a quadword queue. The entries must be octaword-aligned. This function must have write access to header and queue entries.

This function has the following format:

```
int  __PAL_INSQUEQ (void *predecessor, void *new_entry);
```

predecessor

A pointer to an existing entry in the queue.

new_entry

A pointer to the new entry to be inserted.

There are two possible return values:

- 0 if the entry was not the only entry in the queue
- 1 if the entry was the only entry in the queue

6.2.1.65. __PAL_INSQUEQ_D

This function inserts a new entry after an existing entry into a quadword queue deferred. The entries must be octaword-aligned. This function must have write access to header and queue entries.

This function has the following format:

```
int __PAL_INSQUEQ_D (void **predecessor, void *new_entry); /* Deferred */
```

predecessor

A pointer to a pointer to the predecessor entry.

new_entry

A pointer to the new entry to be inserted.

There are two possible return values:

- 0 if the entry was not the only entry in the queue
- 1 if the entry was the only entry in the queue

6.2.1.66. __PAL_LDQP

This function returns the quadword-aligned memory object specified by *address*.

This function has the following format:

```
uint64  __PAL_LDQP (void *address);
```

address

A pointer to the quadword-aligned memory object to be returned.

If the object pointed to by *address* is not quadword-aligned, the result is unpredictable.

6.2.1.67. __PAL_STQP

This function writes the quadword *value* to the memory location pointed to by *address*.

This function has the following format:

```
void  __PAL_STQP (void *address, uint64 value);
```

address

Memory location to be written to.

value

Quadword value to be stored.

If the location pointed to by *address* is not quadword-aligned, the result is unpredictable.

6.2.1.68. __PAL_MFPR_XXXX

These privileged functions return the contents of a particular processor register. The *XXXX* indicates the processor register to be read.

These functions have the following format:

```
unsigned int  __PAL_MFPR_ASTEN (void);  /* AST Enable */
unsigned int  __PAL_MFPR_ASTSR (void);  /* AST Summary Register */
void   *__PAL_MFPR_ESP (void);  /* Executive Stack Pointer */
int     __PAL_MFPR_FEN (void);  /* Floating-Point Enable */
int     __PAL_MFPR_IPL (void);  /* Interrupt Priority Level */
int     __PAL_MFPR_MCES (void); /* Machine Check Error Summary */
void   *__PAL_MFPR_PCBB (void); /* Privileged Context Block Base */
int64   __PAL_MFPR_PRBR (void); /* Processor Base Register */
int     __PAL_MFPR_PTBR (void); /* Page Table Base Register */
void   *__PAL_MFPR_SCBB (void); /* System Control Block Base */
unsigned  int  __PAL_MFPR_SISR (void); /* Software Interrupt Summary
 Register */
void    *__PAL_MFPR_SSP (void); /* Supervisor Stack Pointer */
int64    __PAL_MFPR_TBCHK (void *address); /* Translation Buffer Check */
void    *__PAL_MFPR_USP (void);   /* User Stack Pointer */
void    *__PAL_MFPR_VPTB (void);  /* Virtual Page Table */
int64    __PAL_MFPR_WHAMI (void); /* Who Am I */
```

6.2.1.69. __PAL_MTPR_XXXX

These privileged functions load a value into one of the special processor registers. The *XXXX* indicates the processor register to be loaded.

These functions have the following format:

```
void  __PAL_MTPR_ASTEN (unsigned int mask);  /* AST Enable */
void  __PAL_MTPR_ASTSR (unsigned int mask);  /* AST Summary Register */
void  __PAL_MTPR_DATFX (int value);      /* Data Alignment Trap Fixup */
void  __PAL_MTPR_ESP (void *address);    /* Executive Stack Pointer */
void  __PAL_MTPR_FEN (int value);        /* Floating-Point Enable */
void  __PAL_MTPR_IPIR (int64 number);    /* Interprocessor Interrupt
 Request */
int   __PAL_MTPR_IPL (int value);        /* Interrupt Priority Level */
void  __PAL_MTPR_MCES (int value);       /* Machine Check Error Summary */
void  __PAL_MTPR_PRBR (int64 value);     /* Processor Base Register */
void  __PAL_MTPR_SCBB (void *address);   /* System Control Block Base */
void  __PAL_MTPR_SIRR (int level);       /* Software Interrupt Request
 Register */
void  __PAL_MTPR_SSP (int *address);     /* Supervisor Stack Pointer */
void  __PAL_MTPR_TBIA (void);            /* User Stack Pointer */
void  __PAL_MTPR_TBIAP (void);           /* Translation Buffer Invalidate
 All Process */
void  __PAL_MTPR_TBIS (void *address);   /* Translation Buffer Invalidate
 Single */
```

```
void  __PAL_MTPR_TBISD (void *address);  /* Translation Buffer Invalidate
 Single Data */
void  __PAL_MTPR_TBISI (void *address);  /* Translation Buffer Invalidate
 Single Instruction */
void  __PAL_MTPR_USP (void *address);    /* User Stack Pointer */
void  __PAL_MTPR_VPTB (void *address);   /* Virtual Page Table */
```

6.2.1.70. __PAL_PROBER

This function checks the read accessibility of the first and last byte of the given address and offset pair.

This function has the following format:

```
int  __PAL_PROBER (const void *base_address, int offset, char mode);
```

base_address

The pointer to the memory segment to be tested for read access.

offset

The signed offset to the last byte in the memory segment.

mode

The processor mode used for checking access.

There are two possible return values:

- 0 if one or both bytes are not accessible
- 1 if both bytes are accessible

6.2.1.71. __PAL_PROBEW

This function checks the write accessibility of the first and last byte of the given address and offset pair.

This function has the following format:

```
int  __PAL_PROBEW (const void *base_address, int offset, char mode);
```

base_address

The pointer to the memory segment to be tested for write access.

offset

The signed offset to the last byte in the memory segment.

mode

The processor mode used for checking access.

There are two possible return values:

- 0 if one or both bytes are not accessible
- 1 if both bytes are accessible

6.2.1.72. __PAL_RD_PS

This function returns the Processor Status (PS).

This function has the following format:

```
uint64  __PAL_RD_PS (void);
```

6.2.1.73. __PAL_REMQHIL

This function removes the first entry from a longword queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system. This function must have write access to the header and queue entries.

This function has the following format:

```
int  __PAL_REMQHIL (void *head, void **removed_entry);
/* At head, interlocked */
```

head

A pointer to the queue header. The header must be aligned on a quadword boundary.

removed_entry

A pointer to the address of the entry removed from the queue.

There are four possible return values:

- −1 if the entry cannot be removed because the secondary interlock failed
- 0 if the queue was empty
- 1 if the entry was removed and the queue has remaining entries
- 2 if the entry was removed and the queue is now empty

6.2.1.74. __PAL_REMQHILR

This function removes the first entry from a longword queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system. This function must have write access to the header and queue entries. All parts of the queue must be memory resident.

This function has the following format:

```
int  __PAL_REMQHILR (void *head, void **removed_entry);
/* At head, interlocked resident */
```

head

A pointer to the queue header. The header must be aligned on a quadword boundary.

removed_entry

A pointer to the address of the entry removed from the queue.

There are four possible return values:

- –1 if the entry cannot be removed because the secondary interlock failed
- 0 if the queue was empty
- 1 if the entry was removed and the queue has remaining entries
- 2 if the entry was removed and the queue is now empty

6.2.1.75. __PAL_REMQHIQ

This function removes the first entry from a quadword queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system. This function must have write access to the header and queue entries.

This function has the following format:

```
int  __PAL_REMQHIQ (void *head, void **removed_entry);
/* At head, interlocked */
```

head

A pointer to the queue header. The header must be aligned on an octaword boundary.

removed_entry

A pointer to the address of the entry removed from the queue.

There are four possible return values:

- –1 if the entry cannot be removed because the secondary interlock failed
- 0 if the queue was empty
- 1 if the entry was removed and the queue has remaining entries
- 2 if the entry was removed and the queue is now empty

6.2.1.76. __PAL_REMQHIQR

This function removes the first entry from a quadword queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system. This function must have write access to the header and queue entries. All parts of the queue must be memory resident.

This function has the following format:

```
int  __PAL_REMQHIQR (void *head, void **removed_entry);
/* At head, interlocked resident */
```

head

A pointer to the queue header. The header must be aligned on an octaword boundary.

removed_entry

A pointer to the address of the entry removed from the queue.

There are four possible return values:

- –1 if the entry cannot be removed because the secondary interlock failed
- 0 if the queue was empty
- 1 if the entry was removed and the queue has remaining entries
- 2 if the entry was removed and the queue is now empty

6.2.1.77. __PAL_REMQTIL

This function removes the last entry from a longword queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system. This function must have write access to the header and queue entries.

This function has the following format:

```
int  __PAL_REMQTIL (void *head, void **removed_entry);
/* At tail, interlocked */
```

head

A pointer to the queue header. The header must be aligned on a quadword boundary.

removed_entry

A pointer to the address of the entry removed from the queue.

There are four possible return values:

- –1 if the entry cannot be removed because the secondary interlock failed
- 0 if the queue was empty
- 1 if the entry was removed and the queue has remaining entries
- 2 if the entry was removed and the queue is now empty

6.2.1.78. __PAL_REMQTILR

This function removes the last entry from a longword queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system. This function must have write access to the header and queue entries. All parts of the queue must be memory resident.

This function has the following format:

```
int  __PAL_REMQTILR (void *head, void **removed_entry);
/* At tail, interlocked resident */
```

head

A pointer to the queue header. The header must be aligned on a quadword boundary.

removed_entry

A pointer to the address of the entry removed from the queue.

There are four possible return values:

- –1 if the entry cannot be removed because the secondary interlock failed
- 0 if the queue was empty
- 1 if the entry was removed and the queue has remaining entries
- 2 if the entry was removed and the queue is now empty

6.2.1.79. __PAL_REMQTIQ

This function removes the last entry from a quadword queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system. This function must have write access to the header and queue entries.

This function has the following format:

```
int  __PAL_REMQTIQ (void *head, void **removed_entry);
/* At tail, interlocked */
```

head

A pointer to the queue header. The header must be aligned on an octaword boundary.

removed_entry

A pointer to the address of the entry removed from the queue.

There are four possible return values:

- –1 if the entry cannot be removed because the secondary interlock failed
- 0 if the queue was empty
- 1 if the entry was removed and the queue has remaining entries
- 2 if the entry was removed and the queue is now empty

6.2.1.80. __PAL_REMQTIQR

This function removes the last entry from a quadword queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system. This function must have write access to the header and queue entries. All parts of the queue must be memory resident.

This function has the following format:

```
int  __PAL_REMQTIQR (void *head, void **removed_entry);
/* At tail, interlocked resident */
```

head

A pointer to the queue header. The header must be aligned on an octaword boundary.

removed_entry

A pointer to the address of the entry removed from the queue.

There are four possible return values:

- –1 if the entry cannot be removed because the secondary interlock failed
- 0 if the queue was empty
- 1 if the entry was removed and the queue has remaining entries
- 2 if the entry was removed and the queue is now empty

6.2.1.81. __PAL_REMQUEL

This function removes an entry from a longword queue. This function must have write access to header and queue entries.

This function has the following format:

```
int  _PAL_REMQUEL (void *entry, void **removed_entry);
```

entry

A pointer to the queue entry to be removed.

removed_entry

A pointer to the address of the entry removed from the queue.

There are three possible return values:

- –1 if the queue was empty
- 0 if the entry was removed and the queue is now empty
- 1 if the entry was removed and the queue has remaining entries

6.2.1.82. __PAL_REMQUEL_D

This function removes an entry from a longword queue deferred. This function must have write access to header and queue entries.

This function has the following format:

```
int  __PAL_REMQUEL_D (void **entry, void **removed_entry);  /* Deferred */
```

entry

A pointer to a pointer to the queue entry to be removed.

removed_entry

A pointer to the address of the entry removed from the queue.

There are three possible return values:

- –1 if the queue was empty
- 0 if the entry was removed and the queue is now empty
- 1 if the entry was removed and the queue has remaining entries

6.2.1.83. __PAL_REMQUEQ

This function removes an entry from a quadword queue. This function must have write access to header and queue entries.

This function has the following format:

```
int  __PAL_REMQUEQ (void *entry, void **removed_entry);
```

entry

A pointer to the queue entry to be removed.

removed_entry

A pointer to the address of the entry removed from the queue.

There are three possible return values:

- –1 if the queue was empty
- 0 if the entry was removed and the queue is now empty
- 1 if the entry was removed and the queue has remaining entries

6.2.1.84. __PAL_REMQUEQ_D

This function removes an entry from a quadword queue deferred. This function must have write access to header and queue entries.

This function has the following format:

```
int  __PAL_REMQUEQ_D (void **entry, void **removed_entry);  /* Deferred */
```

entry

A pointer to a pointer to the queue entry to be removed.

removed_entry

A pointer to the address of the entry removed from the queue.

There are three possible return values:

- –1 if the queue was empty
- 0 if the entry was removed and the queue is now empty
- 1 if the entry was removed and the queue has remaining entries

6.2.1.85. __PAL_SWPCTX

This function returns ownership of the data structure that contains the current hardware privileged context (the HWPCB) to the operating system and passes ownership of the new HWPCB to the processor.

This function has the following format:

```
void  __PAL_SWPCTX (void *address);
```

address

A pointer to the new HWPCB.

6.2.1.86. __PAL_SWASTEN

This function swaps the previous state of the Asynchronous System Trap (AST) enable bit for the new state. The new state is supplied in bit 0 of *new_state_mask*. The previous state is returned, zero-extended.

A check is made to determine if an AST is pending. If the enabling conditions are present for an AST at the completion of this instruction, the AST occurs before the next instruction.

This function has the following format:

```
unsigned int  __PAL_SWASTEN (int new_state_mask);
```

new_state_mask

An integer whose 0 bit is the new state of the AST enable bit.

6.2.1.87. __PAL_WR_PS_SW

This function writes the low-order three bits of *mask* into the Processor Status software field (PS<SW>).

This function has the following format:

```
void  __PAL_WR_PS_SW (int mask);
```

mask

An integer whose low-order three bits are written into PS<SW>.

6.2.1.88. _popcnt

The _popcnt built-in function returns the number of "1" bits (0 to 64) in its argument. For example, _popcnt(12) returns 2.

This function has the following format:

```
int64 _popcnt (unsigned int64);
```

6.2.1.89. _poppar

The _poppar built-in function returns 1 if the number of "1" bits in its argument is odd; otherwise it returns 0. For example, _poppar(12) returns 0.

This function has the following format:

```
int64 _poppar (unsigned int64);
```

6.2.1.90. Read Process Cycle Counter (__RPCC)

The __RPCC function reads the current process cycle counter.

This function has the following format:

```
uint64  __RPCC (void);
```

6.2.1.91. Sine (__SIN)

The __SIN built-in is functionally equivalent to its counterpart, `sin`, in the standard header file `<math.h>`.

Its format is also the same:

```
#include <math.h>
double  __SIN (double x);
```

x

A radian value.

This built-in offers performance improvements because there is less call overhead associated with its use.

If you include `<math.h>`, the built-in is automatically used for all occurrences of `sin`. To disable the built-in, use `#undef sin`.

6.2.1.92. Single-Precision, Floating-Point Arithmetic Built-in Functions

The following built-in functions provide single-precision, floating-point chopped arithmetic:

__ADDF_C	__ADDS_C	__SUBF_C	__SUBS_C
__MULF_C	__MULS_C	__DIVF_C	__DIVS_C

They have the following format:

```
float  __op{F,S}_C (float operand1, float operand2);
```

Where *op* is one of ADD, SUB, MUL, DIV, and {F,S} represents VAX or IEEE floating-point arithmetic, respectively.

The result of the arithmetic operation is returned.

6.2.1.93. Test for Bit Clear then Clear Bit Interlocked (__INTERLOCKED_TESTBITCC_QUAD)

The __INTERLOCKED_TESTBITCC_QUAD function performs the following functions in interlocked fashion:

1. Returns the complement of the specified bit before being cleared.
2. Clears the bit.

This function has the following formats:

```
int  __INTERLOCKED_TESTBITCC_QUAD (volatile void *address,
int bit_position);

int  __INTERLOCKED_TESTBITCC_QUAD_RETRY (volatile void *address,
```

```
int bit_position, int retry, int *status);
```

address

The quadword-aligned base address of the bit field.

bit_position

The position within the field of the bit that you want cleared, in the range of 0 to 63.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the quadword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

6.2.1.94. Test for Bit Clear then Clear Bit Interlocked (__TESTBITCCI)

The __TESTBITCCI function performs the following operations in interlocked fashion:

- Returns the complement of the specified bit before being cleared
- Clears the bit

This function has the following format:

```
int  __TESTBITCCI (void *address, int position, ...);
```

address

The base address of the field.

position

The position within the field of the bit that you want cleared.

...

An optional retry count of type `int`. If specified, the retry count indicates the number of times the operation is attempted (which will be at least once, even if the count argument is 0).

6.2.1.95. Test for Bit Set Then Set Bit Interlocked (__INTERLOCKED_TESTBITSS_QUAD)

The __INTERLOCKED_TESTBITSS_QUAD function performs the following functions in interlocked fashion:

1. Returns the value of the specified bit before being set.
2. Sets the bit.

This function has the following formats:

```
int  __INTERLOCKED_TESTBITSS_QUAD (volatile void *address,
int bit_position);

int  __INTERLOCKED_TESTBITSS_QUAD_RETRY (volatile void *address,
int expression, int retry, int *status);
```

address

The quadword-aligned base address of the bit field.

bit_position

The position within the field of the bit that you want cleared, in the range of 0 to 63.

retry

A retry count of type `int` that indicates the number of times the operation is attempted (which is at least once, even if the *retry* argument is 0). If the operation cannot be performed successfully in the specified number of retries, the function returns without updating the longword.

status

A pointer to an integer that is set to 0 if the operation did not succeed within the specified number of retries, and set to 1 if the operation succeeded.

6.2.1.96. Test for Bit Set then Set Bit Interlocked (__TESTBITSSI)

The __TESTBITSSI function performs the following operations in interlocked fashion:

- Returns the value of the specified bit before being set
- Sets the bit

This function has the following format:

```
int  __TESTBITSSI (void *address, int position, ...);
```

address

The base address of the field.

position

The position within the field of the bit that you want set.

…

An optional retry count of type `int`. If specified, the retry count indicates the number of times the operation is attempted (which will be at least once, even if the count argument is 0).

6.2.1.97. _trailz

The _trailz built-in function returns the number of trailing zeros (counting after the least significant set bit to the least significant bit position) in its argument. For example, _trailz(2) returns 1, and _trailz(0) returns 64.

This function has the following format:

```
int64 _trailz (unsigned int64);
```

6.2.1.98. Trap Barrier Instruction (__TRAPB)

The __TRAPB function allows software to guarantee that, in a pipeline implementation, all previous arithmetic instructions will be completed without incurring any arithmetic traps before any instructions after the TRAPB instruction are issued.

This function has the following format:

```
void  __TRAPB (void);
```

6.2.1.99. Unsigned Quadword Multiply High (__UMULH)

The __UMULH function performs a quadword multiply high instruction.

This function has the following format:

```
uint64  __UMULH (uint64 operand1, uint64 operand2);
```

operand1

A 64-bit unsigned integer.

operand2

A 64-bit unsigned integer.

The two operands are multiplied as unsigned integers to produce a 128-bit result. The high-order 64 bits are returned. Note that `uint64` is a `typedef` for the Alpha data type `unsigned __int64`.

6.2.2. Built-In Functions for IA-64 systems (I64 only)

The VSI C built-in functions available on OpenVMS Alpha systems are also available on IA-64 systems, with some differences, as described in this section. This section also describes built-in functions that are specific to IA-64 systems.

6.2.2.1. Builtin Differences on IA-64 Systems

The `<builtins.h>` header file contains comments noting which built-in functions are not available or are not the preferred form for IA-64 systems. The compiler issues diagnostics where using a different built-in function for IA-64 systems would be preferable.

Note

The comments in `<builtins.h>` reflect only what is explicitly present in that header file itself, and in the compiler implementation. You should also consult the content and comments in `<pal_builtins.h>` to determine more accurately what functionality is effectively provided by including `<builtins.h>`. For example, if a program explicitly declares one of the Alpha built-in functions and invokes it without having included `<builtins.h>`, the compiler might issue the BIFNOTAVAIL error message, regardless of whether or not the function is available through a system service. If the compilation does include `<builtins.h>`, and BIFNOTAVAIL is issued, then either there is no support at all for the built-in function or a new version of `<pal_builtins.h>` is needed.

Here is a summary of these differences on IA-64 systems:

- There is no support for the `asm`, `fasm`, and `dasm` intrinsics (declared in the `<c_asm.h>` header file).
- The functionality provided by the special-case treatment of R26 in an Alpha system `asm`, as in `asm("MOV R26,R0")`, is provided by a new built-in function for IA-64 systems:

  ```
  __int64 __RETURN_ADDRESS(void);
  ```

  This built-in function produces the address to which the function containing the built-in call will return (the value of R26 on entry to the function on Alpha systems; the value of B0 on entry to the function on IA-64 systems). This built-in function cannot be used within a function specified to use nonstandard linkage.
- The only PAL function calls implemented as built-in functions within the compiler are the 24 queue-manipulation builtins. The queue manipulation builtins generate calls to new OpenVMS system services SYS$<name>, where <name> is the name of the builtin with the leading underscores removed.

  Any other OpenVMS PAL calls are supported through macros defined in the `<pal_builtins.h>` header file included in the `<builtins.h>` header file. Typically, the macros in `<pal_builtins.h>` transform an invocation of an Alpha system builtin into a call to a system service that performs the equivalent function on an IA-64 system. Two notable exceptions are __PAL_GENTRAP and __PAL_BUGCHK, which instead invoke the I64 specific compiler builtin __break2.
- There is no support for the various floating-point built-in functions used by the OpenVMS math library (for example, operations with chopped rounding and conversions).
- For most built-in functions that take a retry count, the compiler issues a warning message, evaluates the count for possible side effects, ignores it, and then invokes the same function without a retry count. This is necessary because the retry behavior allowed by Alpha load-locked/store-conditional sequences does not exist on IA-64 systems. There are two exceptions to this: __LOCK_LONG_RETRY and __ACQUIRE_SEM_LONG_RETRY; in these cases, the retry behavior involves comparisons of data values, not just load-locked/store-conditional.
- The __CMP_STORE_LONG and __CMP_STORE_QUAD built-in functions produce either a warning or an error, depending on whether or not the compiler can determine if the source and destination addresses are identical. If the addresses are identical, the compiler treats the builtin as the new __CMP_SWAP_ form and issues a warning. Otherwise it is an error.

6.2.2.2. Built-in Functions Specific to IA-64 systems

The `<builtins.h>` header file contains a section at the top conditionalized to just __ia64 with the support for built-in functions specific to IA-64 systems. This includes macro definitions for all of the registers that can be specified to the __getReg, __setReg, __getIndReg, and __setIndReg built-in functions. Parameters that are `const`-qualified require an argument that is a compile-time constant.

The following sections describe the VSI C built-in functions available on OpenVMS IA-64 systems.

6.2.2.3. Get Hardware Register Value (__getReg)

The __getReg function gets the value from a hardware register based on the register index specified. This function produces a corresponding `mov = r` instruction.

This function has the following format:

```
unsigned __int64  __getReg (const int whichReg);
```

whichReg

The index of the hardware register from which the value is obtained. The __getReg and __setReg functions can access the following registers:

```
Register Name         whichReg
_IA64_REG_IP            1016
_IA64_REG_PSR           1019
_IA64_REG_PSR_L         1019

General Integer Registers:

Register Name         whichReg
_IA64_REG_GP            1025
_IA64_REG_SP            1036
_IA64_REG_TP            1037


Application Registers:

Register Name         whichReg
_IA64_REG_AR_KR0        3072
_IA64_REG_AR_KR1        3073

_IA64_REG_AR_KR2        3074
_IA64_REG_AR_KR3        3075
_IA64_REG_AR_KR4        3076
_IA64_REG_AR_KR5        3077
_IA64_REG_AR_KR6        3078
_IA64_REG_AR_KR7        3079
_IA64_REG_AR_RSC        3088
_IA64_REG_AR_BSP        3089
_IA64_REG_AR_BSPSTORE   3090
_IA64_REG_AR_RNAT       3091
_IA64_REG_AR_FCR        3093
_IA64_REG_AR_EFLAG      3096
_IA64_REG_AR_CSD        3097
_IA64_REG_AR_SSD        3098
_IA64_REG_AR_CFLAG      3099
_IA64_REG_AR_FSR        3100
_IA64_REG_AR_FIR        3101
_IA64_REG_AR_FDR        3102
_IA64_REG_AR_CCV        3104
_IA64_REG_AR_UNAT       3108
_IA64_REG_AR_FPSR       3112
_IA64_REG_AR_ITC        3116
_IA64_REG_AR_PFS        3136
_IA64_REG_AR_LC         3137
_IA64_REG_AR_EC         3138


Control Registers:

Register Name         whichReg
```

```
_IA64_REG_CR_DCR        4096
_IA64_REG_CR_ITM        4097

_IA64_REG_CR_IVA        4098
_IA64_REG_CR_PTA        4104
_IA64_REG_CR_IPSR       4112
_IA64_REG_CR_ISR        4113

_IA64_REG_CR_IIP        4115
_IA64_REG_CR_IFA        4116
_IA64_REG_CR_ITIR       4117
_IA64_REG_CR_IIPA       4118

_IA64_REG_CR_IFS        4119
_IA64_REG_CR_IIM        4120
_IA64_REG_CR_IHA        4121
_IA64_REG_CR_LID        4160

_IA64_REG_CR_IVR        4161 *
_IA64_REG_CR_TPR        4162
_IA64_REG_CR_EOI        4163
_IA64_REG_CR_IRR0       4164 *

_IA64_REG_CR_IRR1       4165 *
_IA64_REG_CR_IRR2       4166 *
_IA64_REG_CR_IRR3       4167 *
_IA64_REG_CR_ITV        4168

_IA64_REG_CR_PMV        4169
_IA64_REG_CR_CMCV       4170
_IA64_REG_CR_LRR0       4176
_IA64_REG_CR_LRR1       4177

* getReg only
```

6.2.2.4. Set Hardware Register Value (__setReg)

The __setReg function sets the value for a hardware register based on the register index specified. This function produces a corresponding `mov = r` instruction.

This function has the following format:

```
void  __setReg (const int whichReg, unsigned __int64 value);
```

whichReg

The index of the hardware register whose value is being set. See the __getReg functions for the list of registers that can be accessed.

value

The value to which the register is set.

6.2.2.5. Get Index Register Value (__getIndReg)

The __getIndReg function returns the value of an indexed register. The function accesses a register (*index*) in a register file (*whichIndReg*) of 64-bit registers.

This function has the following format:

```
unsigned __int64  __getIndReg (const int whichIndReg, __int64 index);
```

whichIndReg

The register file.

index

The index in the register file of the hardware register whose value is being requested. See the __getReg functions for the list of registers that can be accessed.

Indirect Registers for getIndReg and setIndReg:

```
Register Name          whichReg

_IA64_REG_INDR_CPUID    9000 *
_IA64_REG_INDR_DBR      9001
_IA64_REG_INDR_IBR      9002
_IA64_REG_INDR_PKR      9003
_IA64_REG_INDR_PMC      9004
_IA64_REG_INDR_PMD      9005
_IA64_REG_INDR_RR       9006
_IA64_REG_INDR_RESERVED 9007

* getIndReg only
```

6.2.2.6. Set Index Register Value (__setIndReg)

The __setIndReg function copies a value into an indexed register. The function accesses a register (*index*) in a register file (*whichIndReg*) of 64-bit registers.

This function has the following format:

```
void  __setIndReg (const int whichIndReg, __int64 index,
unsigned __int64 value);
```

whichIndReg

The register file.

index

The index in the register file of the hardware register to be set. See the __getIndReg function for the list of registers that can be accessed.

value

The value to which the register is set.

6.2.2.7. Generate Break Instruction (__break)

The __break function generates a break instruction with an immediate.

This function has the following format:

```
void  __break (const int __break_arg);
```

__break_arg

An immediate value for the __break instruction to use.

6.2.2.8. Serialize Data (__dsrlz)

The __dsrlz function serializes data. Maps to the srlz.d instruction.

This function has the following format:

```
void  __dsrlz (void);
```

6.2.2.9. Flush Cache Instruction (__fc)

The __fc function flushes a cache line associated with the address given by the argument. Maps to the fcr instruction.

This function has the following format:

```
void  __fc (__int64 __address);
```

__address

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

6.2.2.10. Flush Write Buffers (__fwb)

The __fwb function flushes the write buffers. Maps to the fwb instruction.

This function has the following format:

```
void  __fwb (void);
```

6.2.2.11. Invalidate ALAT (__invalat)

The __invalat function invalidates ALAT. Maps to the invala instruction.

This function has the following format:

```
void  __invalat (void);
```

6.2.2.12. Invalidate ALAT (__invala)

The __invala function is the same as the __invalat function.

6.2.2.13. Execute Serialize (__isrlz)

The __isrlz function executes the serialize instruction. Maps to the srlz.i instruction.

This function has the following format:

```
void  __isrlz (void);
```

6.2.2.14. Insert Data Address Translation Cache (__itcd)

The __itcd function inserts an entry into the data translation cache. Maps to the itc.d instruction.

This function has the following format:

```
void  __itcd (__int64 pa);
```

pa

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

6.2.2.15. Insert Instruction Address Translation Cache (__itci)

The __itci function inserts an entry into the instruction translation cache. Maps to the itc.i instruction.

This function has the following format:

```
void  __itci (__int64 pa);
```

pa

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

6.2.2.16. Insert Data Translation Register (__itrd)

The __itrd function maps to the itr.d instruction.

This function has the following format:

```
void  __itrd (__int64 whichTransReg, __int64 pa);
```

whichTransReg

The data translation register to be used by the itr.d instruction.

pa

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

6.2.2.17. Insert Instruction Translation Register (__itri)

The __itri function maps to the itr.i instruction.

This function has the following format:

```
void  __itri (__int64 whichTransReg, __int64 pa);
```

whichTransReg

The data translation register to be used by the itr.i instruction.

pa

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

6.2.2.18. Purge Translation Cache Entry (__ptce)

The __ptce function maps to the ptc.e instruction.

This function has the following format:

```
void  __ptce (__int64 va);
```

va

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

6.2.2.19. Purge Global Translation Cache (__ptcg)

The __ptcg function purges the global translation cache. Maps to the ptc.g

r,r

instruction.

This function has the following format:

```
void  __ptcg (__int64 va, __int64 pagesz);
```

va

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

pagesz

The address range of the purge.

6.2.2.20. Purge Local Translation Cache (__ptcl)

The __ptcl function purges the local translation cache. Maps to the ptc.l

r,r

instruction.

This function has the following format:

```
void  __ptcl (__int64 va, __int64 pagesz);
```

va

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

pagesz

The address range of the purge.

6.2.2.21. Purge Global Translation Cache and ALAT (__ptcga)

The __ptcga function purges the global translation cache and ALAT. Maps to the ptc.ga r,r instruction.

This function has the following format:

```
void  __ptcga (__int64 va, __int64 pagesz);
```

va

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

pagesz

The address range of the purge.

6.2.2.22. Purge Data Translation Register (__ptrd)

The __ptrd function purges the data translation register. Maps to the ptr.d r,r instruction.

This function has the following format:

```
void  __ptrd (__int64 va, __int64 pagesz);
```

va

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

pagesz

The address range of the purge.

6.2.2.23. Purge Instruction Translation Register (__ptri)

The __ptri function purges the instruction translation register. Maps to the ptr.i

r,r

instruction.

This function has the following format:

```
void  __ptri (__int64 va, __int64 pagesz);
```

va

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

pagesz

The address range of the purge.

6.2.2.24. Reset System Mask (__rsm)

The __rsm function resets the system mask bits of the PSR. Maps to the rsm imm24 instruction.

This function has the following format:

```
void  __rsm (int mask);
```

mask

An integer value inserted into the instruction as a 24-bit immediate value.

6.2.2.25. Reset User Mask (__rum)

The __rum function resets the user mask.

This function has the following format:

```
void  __rum (int mask);
```

mask

An integer value inserted into the instruction as a 24-bit immediate value.

6.2.2.26. Set System Mask (__ssm)

The __ssm function sets the system mask.

This function has the following format:

```
void  __ssm (int mask);
```

mask

An integer value inserted into the instruction as a 24-bit immediate value.

6.2.2.27. Set User Mask (__sum)

The __sum function sets the user mask bits of the PSR. Maps to the sum imm24 instruction.

This function has the following format:

```
void  __sum (int mask);
```

mask

An integer value inserted into the instruction as a 24-bit immediate value.

6.2.2.28. Enable Memory Synchronization (__synci)

The __synci function enables memory synchronization. Maps to the sync.i instruction.

This function has the following format:

```
void  __synci (void);
```

6.2.2.29. Translation Hashed Entry Address (__thash)

The __thash function generates a translation hash entry address. Maps to the thash r = r instruction.

This function has the following format:

```
void  __thash(__int64 __address);
```

__address

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

6.2.2.30. Translation Hashed Entry Tag (__ttag)

The __ttag function generates a translation hash entry tag. Maps to the ttag r=r instruction.

This function has the following format:

```
void  __ttag(__int64 __address);
```

__address

A 64-bit address, as opposed to a 32-bit or 64-bit pointer, that is loaded into a 64-bit general register used by the instruction to be generated.

6.2.2.31. Atomic Compare and Exchange (_InterlockedCompareExchange_acq)

The _InterlockedCompareExchange_acq function atomically compares and exchanges the value specified by the first argument (a 64-bit pointer). This function maps to the cmpxchg4.acq instruction with appropriate setup.

This function has the following format:

```
unsigned __int64  _InterlockedCompareExchange_acq (volatile unsigned int
*Destination, unsigned __int64 Newval, unsigned __int64 Comparand);
```

The value at **Destination* is compared with the value specified by *Comparand*. If they are equal, *Newval* is written to **Destination*, and *Oldval* is returned. The exchange will have taken place if the value returned is equal to the *Comparand*. The following algorithm is used:

```
ar.ccv = Comparand;
Oldval = *Destination;          //Atomic
if (ar.ccv == *Destination)     //Atomic
    *Destination = Newval;      //Atomic
return Oldval;
```

Those parts of the algorithm that are marked "Atomic" are performed atomically by the cmpxchg4.acq instruction. This instruction has acquire ordering semantics; that is, the memory read/write is made visible prior to all subsequent data memory accesses of the *Destination* by other processors.

Destination

The value to be compared with *Comparand* and, if equal, replaced with the value of *Newval*.

Newval

The new value to replace the value in *Destination*.

Comparand

The value with which to compare *Destination*.

6.2.2.32. Atomic Compare and Exchange (_InterlockedCompareExchange64_acq)

The _InterlockedCompareExchange64_acq function is the same as the _InterlockedCompareExchange_acq function, except that those parts of the algorithm that are marked "Atomic" are performed by the cmpxchg8.acq instruction.

This function has the following format:

```
unsigned __int64  _InterlockedCompareExchange64_acq (volatile unsigned
 __int64
*Destination, unsigned __int64 Newval, unsigned __int64 Comparand);
```

6.2.2.33. Atomic Compare and Exchange (_InterlockedCompareExchange_rel)

This function is the same as the _InterlockedCompareExchange_acq function except that those parts of the algorithm that are marked "Atomic" are performed by the cmpxchg4.rel instruction with release ordering semantics; that is, the memory read/write is made visible after all previous memory accesses of the *Destination* by other processors.

This function has the following format:

```
unsigned __int64  _InterlockedCompareExchange_rel (volatile unsigned int
*Destination, unsigned __int64 Newval, unsigned __int64 Comparand);
Atomic Compare and Exchange ( _InterlockedCompareExchange64_rel)
```

6.2.2.34. Atomic Compare and Exchange (_InterlockedCompareExchange64_rel)

This function is the same as the _InterlockedCompareExchange_rel function, except that those parts of the algorithm that are marked "Atomic" are performed by the cmpxchg8.rel instruction.

This function has the following format:

```
unsigned __int64  _InterlockedCompareExchange64_rel (volatile unsigned
 __int64
*Destination, unsigned __int64 Newval, unsigned __int64 Comparand);
```

6.2.2.35. Conditional Atomic Compare and Exchange Longword (__CMP_SWAP_LONG)

The __CMP_SWAP_LONG function performs a conditional atomic compare and exchange operation on a longword. The longword pointed to by *source* is read and compared with the longword *old_value*. If they are equal, the longword *new_value* is written into the longword pointed to by *source*. The read and write is performed atomically, with no intervening access to the same memory region.

The function returns 1 if the write occurs, and 0 otherwise.

This function has the following format:

```
int  __CMP_SWAP_LONG (volatile void *source, int old_value,
 int new_value);
```

source

The longword value to be compared with *old_value*.

old_value

The longword value *source* is compared with.

new_value

The longword value written into *source* if *source* and *old_value* are equal.

6.2.2.36. Conditional Atomic Compare and Exchange Quadword (__CMP_SWAP_QUAD)

The __CMP_SWAP_QUAD function performs a conditional atomic compare and exchange operation on a quadword. The quadword pointed to by *source* is read and compared with the quadword *old_value*. If they are equal, the quadword *new_value* is written into the quadword pointed to by *source*. The read and write is performed atomically, with no intervening access to the same memory region.

The function returns 1 if the write occurs, and 0 otherwise.

This function has the following format:

```
int  __CMP_SWAP_QUAD (volatile void *source, int old_value,
int new_value);
```

source

The quadword value to be compared with *old_value*.

old_value

The quadword value *source* is compared with.

new_value

The quadword value written to *source* if *source* and *old_value* are equal.

6.2.2.37. Conditional Atomic Compare and Exchange Longword with Acquire Semantics (__CMP_SWAP_LONG_ACQ)

The __CMP_SWAP_LONG_ACQ function performs a conditional atomic compare and exchange operation with acquire semantics on a longword. The longword pointed to by *source* is read and compared with the longword *old_value*. If they are equal, the longword *new_value* is written into the longword pointed to by *source*. The read and write is performed atomically, with no intervening access to the same memory region.

Acquire memory ordering guarantees that the memory read/write is made visible *before* all subsequent data accesses to the same memory location by other processors.

The function returns 1 if the write occurs, and 0 otherwise.

This function has the following format:

```
int  __CMP_SWAP_LONG_ACQ (volatile void *source, int old_value,
 int new_value);
```

source

The longword value to be compared with *old_value*.

old_value

The longword value *source* is compared with.

new_value

The longword value written into *source* if *source* and *old_value* are equal.

6.2.2.38. Conditional Atomic Compare and Exchange Quadword with Acquire Semantics (__CMP_SWAP_QUAD_ACQ)

The __CMP_SWAP_QUAD_ACQ function performs a conditional atomic compare and exchange operation with acquire semantics on a quadword. The quadword pointed to by *source* is read and compared with the quadword *old_value*. If they are equal, the quadword *new_value* is written into the quadword pointed to by *source*. The read and write is performed atomically, with no intervening access to the same memory region.

Acquire memory ordering guarantees that the memory read/write is made visible *before* all subsequent memory data accesses to the same memory location by other processors.

The function returns 1 if the write occurs, and 0 otherwise.

This function has the following format:

```
int  __CMP_SWAP_QUAD_ACQ (volatile void *source, int old_value,
int new_value);
```

source

The quadword value to be compared with *old_value*.

old_value

The quadword value *source* is compared with.

new_value

The quadword value written into *source* if *source* and *old_value* are equal.

6.2.2.39. Conditional Atomic Compare and Exchange Longword with Release Semantics (__CMP_SWAP_LONG_REL)

The __CMP_SWAP_LONG_REL function performs a conditional atomic compare and exchange operation with release semantics on a longword. The longword pointed to by *source* is read and compared with the longword *old_value*. If they are equal, the longword *new_value* is written into the longword pointed to by *source*. The read and write is performed atomically, with no intervening access to the same memory region.

Release memory ordering guarantees that the memory read/write is made visible *after* all previous data memory acceses to the same memory location by other processors.

The function returns 1 if the write occurs, and 0 otherwise.

This function has the following format:

```
int  __CMP_SWAP_LONG_REL (volatile void *source, int old_value,
int new_value);
```

source

The longword value to be compared with *old_value*.

old_value

The longword value *source* is compared with.

new_value

The longword value written into *source* if *source* and *old_value* are equal.

6.2.2.40. Conditional Atomic Compare and Exchange Quadword with Release Semantics (__CMP_SWAP_QUAD_REL)

The __CMP_SWAP_QUAD_REL function performs a conditional atomic compare and exchange operation with release semantics on a quadword. The quadword pointed to by *source* is read and compared with the quadword *old_value*. If they are equal, the quadword *new_value* is written into the quadword pointed to by *source*. The read and write is performed atomically, with no intervening access to the same memory region.

Release memory ordering guarantees that the memory read/write is made visible *after* all previous data memory acceses to the same memory location by other processors.

The function returns 1 if the write occurs, and 0 otherwise.

This function has the following format:

```
int  __CMP_SWAP_QUAD_REL (volatile void *source, int old_value,
int new_value);
```

source

The quadword value to be compared with *old_value*.

old_value

The quadword value *source* is compared with.

new_value

The quadword value written into *source* if *source* and *old_value* are equal.

6.2.2.41. Return Address (__RETURN_ADDRESS)

The __RETURN_ADDRESS function produces the address to which the function containing the built-in call will return as a 64-bit integer (on Alpha systems, the value of R26 on entry to the function; on IA-64 systems, the value of B0 on entry to the function).

This built-in function cannot be used within a function specified to use nonstandard linkage.

This function has the following format:

```
__int64  __RETURN_ADDRESS (void);
```

6.2.2.42. Implement Alpha __PAL_GENTRAP and __PAL_BUGCHK Builtins (__break2)

The __break2 function is used to implement the Alpha __PAL_GENTRAP and __PAL_BUGCHK built-in functions on OpenVMS IA-64 systems.

The __break2 function is equivalent to the __break function with the second parameter passed in general register 17:

```
R17 = __R17_value; __break
(__break_code);
```

This function has the following format:

```
void  __break2 (__Integer_Constant __break_code,
unsigned __int64 __r17_value);
```

__breakcode

The particular software condition that has occurred.

__r17_value

The value of R17, a volatile general register reserved by the OpenVMS Itanium calling standard for use by compiled code to communicate with specialized compiler support routines that require out-of-band information passing.

6.2.2.43. Flush Register Stack (__flushrs)

The __flushrs function flushes the register stack.

This function has the following format:

```
void  __flushrs (void);
```

6.2.2.44. Load Register Stack (__loadrs)

The __loadrs function loads the register stack.

This function has the following format:

```
void  __loadrs (void);
```

6.2.2.45. Probe Read-Access Permission (__prober)

The __prober function determines whether read access to the virtual address specified by *__address* bits {60:0} and the region register indexed by *__address* bits {63:61} is permitted at the privilege level given by *__mode* bits {1:0}. It returns 1 if the access is permitted, and 0 otherwise.

This function can probe only with equal or lower privilege levels. If the specified privilege level is higher (lower number), then the probe is performed with the current privilege level.

This function is the same as the Intel __probe_r function.

This function has the following format:

```
int  __prober (__int64 __address, unsigned int __mode);
```

__address

Virtual address for which read-access permission is being checked.

__mode

Privilege level for which read-access permission is being checked.

6.2.2.46. Probe Write-Access Permission (__probew)

The __probew function determines whether write access to the virtual address specified by *__address* bits {60:0} and the region register indexed by *__address* bits {63:61}, is permitted at the privilege level given by *__mode* bits {1:0}. It returns 1 if the access is permitted, and 0 otherwise.

This function can probe only with equal or lower privilege levels. If the specified privilege level is higher (lower number), then the probe is performed with the current privilege level.

This function is the same as the Intel __probe_w function.

This function has the following format:

```
int  __probew (__int64 __address, unsigned int __mode);
```

__address

Virtual address for which write-access permission is being checked.

__mode

Privilege level for which write-access permission is being checked.

6.2.2.47. Translation Access Key (__tak)

The __tak function returns the translation access key.

This function has the following format:

```
unsigned int  __tak (__int64 __address);
```

__address

Virtual address for translation key is being returned.

6.2.2.48. Translate to Physical Address (__tpa)

The __tpa function translates a virtual address to a physical address.

This function has the following format:

```
__int64  __tpa(__int64 __address);
```

__address

Virtual address to be translated.

6.2.3. Built-In Functions for OpenVMS VAX Systems (VAX only)

The following sections describe the VSI C built-in functions available on OpenVMS VAX systems.

The VSI C built-in functions use enumerated `typedefs` to define possible return values. We recommend that you use the enumerated types to store and compare return values.

6.2.3.1. Allocate Bytes from Stack (__ALLOCA)

The __ALLOCA function allocates *n* bytes from the stack.

This function has the following format:

```
void  *__ALLOCA (unsigned int n);
```

n

The number of bytes to be allocated.

A pointer to the allocated memory is returned.

6.2.3.2. Add Aligned Word Interlocked (_ADAWI)

The _ADAWI function adds its source operand to the destination. This function is interlocked against similar operations by other processors or devices in the system.

The _ADAWI function has the following format:

```
typedef enum
{ _adawi_sum_neg=-1,  _adawi_sum_zero,  _adawi_sum_pos}  _ADAWI_STATUS;

_ADAWI_STATUS  _ADAWI (short  __src,  short *__dest);
```

__src

The value to be added to the destination.

__dest

A pointer to the destination. The destination must be aligned on a word boundary. (You can achieve alignment using the `_align` or `__align` storage-class modifier.)

There are three possible return values:

- `adawi_sum_neg` (–1) if the sum when considered to be a signed number is negative
- `adawi_sum_zero` (0) if the sum is 0
- `adawi_sum_pos` (1) if the sum is positive

6.2.3.3. Branch on Bit Clear-Clear Interlocked (_BBCCI)

The _BBCCI function performs the following functions in interlocked fashion:

- Returns the complement of the bit specified by the two arguments
- Clears the bit specified by the two arguments

The _BBCCI function has the following format:

```
typedef enum { _bbcci_oldval_1,  _bbcci_oldval_0}  _BBCCI_STATUS;

_BBCCI_STATUS  _BBCCI (int  __position,  void *__address);
```

__position

The position of the bit within the field.

__address

The base address of the field.

The return value of `_bbcci_oldval_1` (0) or `_bbcci_oldval_0` (1) is the complement of the value of the specified bit before being cleared.

6.2.3.4. Branch on Bit Set-Set Interlocked (_BBSSI)

The _BBSSI function performs the following functions in interlocked fashion:

- Returns the status of the bit specified by the two arguments
- Sets the bit specified by the two arguments

The _BBSSI function has the following format:

```
typedef enum { _bbssi_oldval_0,  _bbcci_oldval_1}  _BBSSI_STATUS;

_BBSSI_STATUS  _BBSSI (int  __position,  void *__address);
```

__position

The position of the bit within the field.

__address

The base address of the field.

The return value of `_bbssi_oldval_0` (0) or `_bbssi_oldval_1` (1) is the value of the specified bit before being set.

6.2.3.5. Find First Clear Bit (_FFC)

The _FFC function finds the position of the first clear bit in a field. The bits are tested for clear status starting at bit 0 and extending to the highest bit in the field.

The _FFC function has the following format:

```
typedef enum { _ff_bit_not_found,  _ff_bit_found}  _FF_STATUS;

_FF_STATUS  _FFC (int  __start,  char  __size,  const void *__base,  int
*__position);
```

__start

The start position of the field.

__size

The size of the field, in bits. The size must be a value from 0 to 32 bits.

__base

The address of the field.

__position

The address of an integer to receive the position of the clear bit. If no bit is clear, the integer is set to the position of the first bit past the last bit tested.

There are two possible return values:

- `_ff_bit_not_found` (0) if all bits in the field are set
- `_ff_bit_found` (1) if a bit with value 0 is found

6.2.3.6. Find First Set Bit (_FFS)

The _FFS function finds the position of the first set bit in a field. The bits are tested for set status starting at bit 0 and extending to the highest bit in the field.

The _FFS function has the following format:

```
typedef enum { _ff_bit_not_found,  _ff_bit_found}  _FF_STATUS;

_FF_STATUS  _FFS (int  __start,  char  __size,  const void *__base, int
*__position);
```

__start

The start position of the field.

__size

The size of the field, in bits. The size must be a value from 0 to 32 bits.

__base

The address of the field.

__position

The address of an integer to receive the position of the set bit. If no bit is set, the integer is set to the position of the first bit past the last bit tested.

There are two possible return values:

- `_ff_bit_not_found` (0) if all bits in the field are clear
- `_ff_bit_found` (1) if a bit with value 1 is found

6.2.3.7. Halt (_HALT)

The _HALT function halts the processor when executed by a process running in kernel mode. This is a privileged function.

The _HALT function has the following format:

```
void    _HALT (void);
```

6.2.3.8. Insert Entry into Queue at Head Interlocked (_INSQHI)

The _INSQHI function inserts an entry into the front of a queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system.

The _INSQHI function has the following format:

```
typedef enum {_insqi_inserted_many,  _insqi_not_inserted,
 _insqi_inserted_only}  _INSQI_STATUS;

_INSQI_STATUS  _INSQHI (void *__new_entry,  void *__head);
```

__new_entry

A pointer to the new entry to be inserted. The entry must be aligned on a quadword boundary. (You can achieve alignment using the `_align` or `__align` storage-class modifier.)

__head

A pointer to the queue header. The header must be aligned on a quadword boundary. (You can achieve alignment using the `_align` or `__align` storage-class modifier.)

There are three possible return values:

- `_insqi_inserted_many` (0) if the entry was inserted, but it was not the only entry in the list
- `_insqi_not_inserted` (1) if the entry was not inserted because the secondary interlock failed
- `_insqi_inserted_only` (2) if the entry was inserted and it was the only entry in the list

6.2.3.9. Insert Entry into Queue at Tail Interlocked (_INSQTI)

The _INSQTI function inserts an entry at the end of a queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system.

The _INSQTI function has the following format:

```
typedef enum {_insqi_inserted_many,  _insqi_not_inserted,
 _insqi_inserted_only}  _INSQI_STATUS;

_INSQI_STATUS  _INSQTI (void *__new_entry,  void *__head);
```

__new_entry

A pointer to the new entry to be inserted. The entry must be aligned on a quadword boundary. (You can achieve alignment using the `_align` or `__align` storage-class modifier.)

__head

A pointer to the queue header. The header must be aligned on a quadword boundary. (You can achieve alignment using the `_align` or `__align` storage-class modifier.)

There are three possible return values:

- `_insqi_inserted_many` (0) if the entry was inserted, but it was not the only entry in the list
- `_insqi_not_inserted` (1) if the entry was not inserted because the secondary interlock failed
- `_insqi_inserted_only` (2) if the entry was inserted and it was the only entry in the list

6.2.3.10. Insert Entry in Queue (_INSQUE)

The _INSQUE function inserts a new entry into a queue following an existing entry.

The _INSQUE function has the following format:

```
typedef enum { _insque_inserted_only,  _insque_inserted_many}
 _INSQUE_STATUS;

_INSQUE_STATUS  _INSQUE (void *__new_entry,  void *__predecessor);
```

__new_entry

A pointer to the new entry to be inserted.

__predecessor

A pointer to an existing entry in the queue.

There are two possible return values:

- `_insque_inserted_only` (0) if the entry was the only entry in the queue
- `_insque_inserted_many` (1) if the entry was not the only entry in the queue

6.2.3.11. Locate Character (_LOCC)

The _LOCC function locates the first character in a string matching the target character.

The _LOCC function has the following format:

```
unsigned short _LOCC (char __target, unsigned short __length, const char
*__string, ...);
```

__target

The character being searched.

__length

The length of the searched string. The length must be a value from 0 to 65,535.

__string

A pointer to the searched string.

...

An optional *position* argument, which is a pointer to a pointer to `char`. If the searched character is found, this output argument is updated to point to the character found. If the character is not found, this argument is set to the address one byte beyond the string.

If the target character is found, the return value is the number of bytes remaining in the string; otherwise, the return value is 0.

6.2.3.12. Move from Processor Register (_MFPR)

The _MFPR function returns the contents of a processor register. This is a privileged function.

The _MFPR function has the following formats:

```
void _MFPR (int register_num, int *destination);
void _MFPR (int register_num, unsigned int *destination);
```

register_num

The number of the privileged register to be read.

destination

A pointer to the location receiving the value from the register. This location can be a `signed` or `unsigned int`.

6.2.3.13. Move Character 3 Operand (_MOVC3)

The _MOVC3 function copies a block of memory.

The _MOVC3 function has the following format:

```
void _MOVC3 (unsigned short __length, const char *__src, char
*__dest, ...);
```

__length

The length of the source string, in bytes. The length must be a value from 0 to 65,535.

__src

A pointer to the source string.

__dest

A pointer to the destination memory.

...

One or two optional arguments:

- *endscr*

  A pointer to a pointer to `char`. The _MOVC3 function sets this output argument to the address of the byte beyond the source string. Although this is an optional argument, it is required if *enddest* is specified.

- *enddest*

A pointer to a pointer to `char`. The _MOVC3 function sets this output argument to the address of the byte beyond the destination string.

6.2.3.14. Move Character 5 Operand (_MOVC5)

The _MOVC5 function allows the source string specified by the pointer and length pair to be moved to the destination string specified by the other pointer and length pair. If the source string is smaller than the destination string, the destination string is padded with the specified character.

The _MOVC5 function has the following format:

```
void  _MOVC5 (unsigned short  __srclen,  const char *__src,
              char __fill, unsigned short  __destlen,  char
              *__dest, ...);
```

__srclen

The length of the source string, in bytes. The length must be a value from 0 to 65,535.

__src

A pointer to the source string.

__fill

The fill character to be used if the source string is smaller than the destination string.

__destlen

The length of the destination string, in bytes. The length must be a value from 0 to 65,535.

__dest

A pointer to the destination string.

...

One to three optional arguments:

- *unmoved_src*

  A pointer to an `unsigned short` integer. The _MOVC5 function sets this output argument to the number of unmoved bytes remaining in the source string. This argument is optional if the *endscr* argument is not specified.

- *endscr*

  A pointer to a pointer to `char`. The _MOVC5 function sets this output argument to the address of the byte beyond the source string. Although this is an optional argument, it is required if *enddest* is specified.

- *enddest*

  A pointer to a pointer to `char`. The _MOVC5 function sets this output argument to the address of the byte beyond the destination string.

6.2.3.15. Move from Processor Status Longword (_MOVPSL)

The _MOVPSL function stores the value of the Processor Status Longword (PSL).

The _MOVPSL function has the following format:

```
void  _MOVPSL (void *__psl);
```

__psl

The address of the location for storing the value of the PSL.

6.2.3.16. Move to Processor Register (_MTPR)

The _MTPR function loads a value into one of the special processor registers. It is a privileged function.

The _MTPR function has the following format:

```
int _MTPR (int src, int register_num);
```

src

The value to store into the processor register.

register_num

The number of a privileged register to be updated.

The return value is the V condition flag from the Processor Status Longword (PSL).

6.2.3.17. Probe Read Accessibility (_PROBER)

The _PROBER function checks to see if you can read the first and last byte of the given address and length pair.

The _PROBER function has the following format:

```
typedef enum   { _probe_not_accessible,  _probe_accessible}  _PROBE_STATUS;

_PROBE_STATUS  _PROBER (char  __mode,  unsigned short  __length,  const
 void *__address);
```

__mode

The processor mode used for checking the access.

__length

The length of the memory segment, in bytes. The length must be a value from 0 to 65,535.

On OpenVMS Alpha systems, this parameter is the offset to the last byte in the memory segment, and not the memory segment length.

__address

The pointer to the memory segment to be tested for read access.

There are two possible return values:

- `_probe_not_accessible` (0) if one or both bytes are not accessible
- `_probe_accessible` (1) if both bytes are accessible

6.2.3.18. Probe Write Accessibility (_PROBEW)

The _PROBEW function checks the write accessibility of the first and last byte of the given address and length pair.

The _PROBEW function has the following format:

```
typedef enum   { _probe_not_accessible,  _probe_accessible}  _PROBE_STATUS;

_PROBE_STATUS  _PROBEW (char  __mode,  unsigned short  __length,  const
 void *__address);
```

__mode

The processor mode used for checking the access.

__length

On OpenVMS VAX systems, the length of the memory segment, in bytes. The length must be a value from 0 to 65,535.

On OpenVMS Alpha systems, this parameter is the offset to the last byte in the memory segment, and not the memory segment length.

__address

The pointer to the memory segment to be tested for write access.

There are two possible return values:

- `_probe_not_accessible` (0) if one or both bytes are not accessible
- `_probe_accessible` (1) if both bytes are accessible

6.2.3.19. Read General-Purpose Register (_READ_GPR)

The _READ_GPR function returns the value of a general-purpose register.

The _READ_GPR function has the following format:

```
int _READ_GPR (int register_num);
```

register_num

An integer constant expression giving the number of the general-purpose register to be read.

The return value is the value of the general-purpose register.

6.2.3.20. Remove Entry from Queue at Head Interlocked (_REMQHI)

The _REMQHI function removes the first entry from the queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system.

The _REMQHI function has the following format:

```
typedef enum   { _remqi_removed_more, _remqi_not_removed,
                 _remqi_removed_empty, _remqi_empty}  _REMQI_STATUS;

_REMQI_STATUS  _REMQHI (void *__head,  void *__removed_entry);
```

__head

A pointer to the queue header. The header must be aligned on a quadword boundary. (You can achieve alignment using the `_align` or `__align` storage-class modifier.)

__removed_entry

A pointer that _REMQHI sets to point to the removed entry.

There are four possible return values:

- `_remqi_removed_more` (0) if the entry was removed and the queue has remaining entries
- `_remqi_not_removed` (1) if the entry could not be removed because the secondary interlock failed
- `_remqi_removed_empty` (2) if the entry was removed and the queue is now empty
- `_remqi_empty` (3) if the queue was empty

6.2.3.21. Remove Entry from Queue at Tail Interlocked (_REMQTI)

The _REMQTI function removes the last entry from the queue in an indivisible manner. This operation is interlocked against similar operations by other processors or devices in the system.

The _REMQTI function has the following format:

```
typedef enum   { _remqi_removed_more,  _remqi_not_removed,
                 _remqi_removed_empty, _remqi_empty}    _REMQI_STATUS;

_REMQI_STATUS  _REMQTI (void *__head,  void *__removed_entry);
```

__head

A pointer to the queue header. The header must be aligned on a quadword boundary. (You can achieve alignment using the `_align` or `__align` storage-class modifier.)

__removed_entry

A pointer to a pointer that _REMQTI sets to point to the removed entry.

There are four possible return values:

- `_remqi_removed_more` (0) if the entry was removed and the queue has remaining entries
- `_remqi_not_removed` (1) if the entry could not be removed because the secondary interlock failed
- `_remqi_removed_empty` (2) if the entry was removed and the queue is now empty

- `_remqi_empty` (3) if the queue was empty

6.2.3.22. Remove Entry from Queue (_REMQUE)

The _REMQUE function removes an entry from a queue.

The _REMQUE function has the following format:

```
typedef enum   { _remque_removed_more,    _remque_removed_empty,
                 _remque_empty}    _REMQUE_STATUS;

_REMQUE_STATUS  _REMQUE (void *__entry,  void *__removed_entry);
```

__entry

A pointer to the queue entry to be removed.

__removed_entry

A pointer to a pointer that _REMQUE sets to the address of the entry removed from the queue.

There are three possible return values:

- `_remque_removed_more` (0) if the entry was removed and the queue has remaining entries
- `_remque_removed_empty` (1) if the entry was removed and the queue is now empty
- `_remque_empty` (2) if the queue was empty

6.2.3.23. Scan Characters (_SCANC)

The _SCANC function locates the first character in a string with the desired attributes. The attributes are specified through a table and a mask.

The _SCANC function has the following format:

```
unsigned short  _SCANC (unsigned short  __length,  const char *__string,
                        const char *__table, char __mask, ...);
```

__length

The length of the string to be scanned, in bytes. The length must be a value from 0 to 65,535.

__string

A pointer to the string to be scanned.

__table

A pointer to the table.

__mask

The mask.

...

An optional *match* argument, which is a pointer to a pointer to `char`. The _SCANC function sets this output argument to the address of the byte that matched. (If no match occurs, this argument is set to the address of the byte following the string.)

The return value is the number of bytes remaining in the string if a match was found; otherwise, the return value is 0.

6.2.3.24. Skip Character (_SKPC)

The _SKPC function locates the first character in a string that does not match the target character.

The _SKPC function has the following format:

```
unsigned short  _SKPC (char  __target,  unsigned short  __length,
                       const char *__string, ... );
```

__target

The target character.

__length

The length of the string, in bytes. The length must be a value from 0 to 65,535.

__string

A pointer to the string to be scanned.

...

An optional *position* argument, which is a pointer to a pointer to `char`. The _SKPC function sets this output argument to the address of the nonmatching character. (If all the characters in the string match, this argument is set to the address of the first byte beyond the string.)

The return value is the number of bytes remaining in the string if an unequal byte was located; otherwise, the return value is 0.

6.2.3.25. Span Characters (_SPANC)

The _SPANC function locates the first character in a string without certain attributes. The attributes are specified through a table and a mask.

The _SPANC function has the following format:

```
unsigned short  _SPANC (unsigned short  __length,  const char *__string,
                        const char *__table, char __mask, ...);
```

__length

The length of the string, in bytes. The length must be a value from 0 to 65,535.

__string

A pointer. It points to the string to be scanned.

__table

A pointer to the table.

__mask

The mask.

...

An optional *position* argument, which is a pointer to a pointer to `char`. The _SPANC function sets this output argument to the address of the nonmatching character. (If all the characters in the string match, this argument is set to the address of the first byte beyond the string.)

The return value is the number of bytes remaining in the string if a match was found; otherwise, the return value is 0.

Appendix A. Migrating from VAX C

This appendix documents many features that distinguish VSI C for OpenVMS Systems from VAX C Version 3.2.

This appendix was written for the first ANSI C Standard conforming release of the VSI C compiler as a guide for installations migrating from VAX C to VSI C. It is not intended as a compendium of new features for all VSI C versions. For a summary of new features for the current version of the compiler, see the release notes and the New and Changed Features section in the Preface of this and the other VSI C manuals.

The major focus of VSI C for OpenVMS Systems is to bring it into full conformance with the C Standard. The language described by the C Standard differs in many ways from the language originally implemented by VAX C. These differences include additional language features and constructs, the removal of obsolete features and usages, and a number of other changes that generally involve a tightening up of semantic rules.

Some of the new C standard features have already been implemented in previous versions of VAX C. Some of these are: support for function prototypes, the `const` and `volatile` type qualifiers, and the `void` type specifier.

Although every attempt has been made to maintain compatibility with earlier versions of the VAX C compiler, many of the changes required to bring the compiler into conformance with the C Standard would introduce unavoidable incompatibilities with these earlier versions. For example, VAX C supports a number of language and semantic extensions that are not standard-conformant.

Therefore, to provide compatibility with previous versions of the compiler, VSI C for OpenVMS Systems supports several modes of operation:

- Strict ANSI C Standard mode, in which all nonstandard constructs and usages (including VAX C extensions) are diagnosed
- Relaxed mode (the default on OpenVMS systems), in which the compiler follows the ANSI C standard but also accepts additional VSI keywords and predefined macros that do not begin with an underscore
- VAX C mode, in which as many previously supported features as possible continue to be supported
- Common mode, in which extensions to the ULTRIX portable C compiler (`pcc`) are supported

Note that some of the language changes dictated by the C Standard are present in VAX C mode. Some of these changes are quiet changes; that is, they cannot be detected as such by the compiler, so no diagnostic messages are issued. Also note that some extensions are permitted in the strict ANSI C mode. These extensions are diagnosed, but with no greater severity than Warning. Both types of changes are included in the following sections that describe all new and changed features.

A.1. Features Affecting the Compiler

This section describes VSI C compiler features. (*Section A.2, "Features Affecting the VSI C Run-Time Library and Include Files"* describes features that affect the VSI C run-time library and include files.)

A.1.1. VSI C Qualifiers

Qualifiers new to VSI C:

- /[NO]ANSI_ALIAS *(Alpha only)*—Directs the compiler to assume the ANSI C aliasing rules. By so doing, the compiler has the freedom to generate better optimized code.
- /ASSUME=(option,...)—Controls compiler assumptions.
- /DECC—Invokes the VSI C compiler. For OpenVMS VAX systems, the default is set to either /DECC or /VAXC during installation.

  For OpenVMS Alpha systems, specifying /DECC is equivalent to not specifying it; it is supported to provide compatibility with VSI C on OpenVMS VAX systems.
- /ENDIAN=option (Alpha only)—Controls whether big or little endian ordering of bytes is carried out in character constants.
- /EXTERN_MODEL—In conjunction with the /[NO]SHARE_GLOBALS qualifier, controls the initial extern model of the compiler. Also see `#pragma extern_model`.
- /FLOAT—Controls the format of floating-point variables. It replaces the /[NO]G_FLOAT qualifier, which is retained for compatibility.
- /GRANULARITY (Alpha only)—Determines how much memory to effectively cache for memory reference, by the combination of the compiler and the underlying system.
- /IEEE_MODE=option (Alpha only)—Selects the IEEE floating-point mode to be used if /FLOAT=IEEE_FLOAT is specified.
- /L_DOUBLE_SIZE=option (Alpha only)—Determines how the compiler interprets the `long double` type.
- /[NO]MEMBER_ALIGNMENT—Controls alignment of data structure members. For OpenVMS Alpha systems, the default is /MEMBER_ALIGNMENT, which aligns structure members on the next boundary appropriate to the type of the member. For OpenVMS VAX systems, the default is /NOMEMBER_ALIGNMENT, which aligns structure members on byte boundaries.
- /NAMES—Converts all definitions and references of external symbols and psects to the specified case (UPPERCASE or AS_IS).
- /NESTED_INCLUDE_DIRECTORY[=option]—Controls the directories that the compiler searches when looking for nested include files that are included using the quoted form of the `#include` preprocessor directive.
- /OPTIMIZE—Determines whether VSI C performs various code optimizations.
- /[NO]PLUS_LIST_OPTIMIZE (Alpha only)—Provides improved optimization and code generation across file boundaries that would not be available if the files were compiled separately.
- /[NO]PREFIX_LIBRARY_ENTRIES=(option,...)—Controls the VSI C Run-Time LIbrary (RTL) name prefixing.
- /REENTRANCY[=option] (Alpha only)—Controls the type of reentrancy that reentrant VSI C RTL routines will exhibit. (See also the `decc$set_reentrancy` RTL routine.)
- /ROUNDING_MODE=option (Alpha only)—Lets you select an IEEE rounding mode, if /FLOAT=IEEE_MODE is specified.

- /[NO]SHARE_GLOBALS—Specifies whether external objects are to be marked share or noshare. Used in conjunction with /EXTERN_MODEL to control the initial extern model of the compiler. Also see the `#pragma extern_model` preprocessor directive.
- /[NO]STANDARD—Enhanced to include the following options (in addition to /STANDARD=PORTABLE):
  - ANSI89
  - RELAXED
  - COMMON
  - VAXC
  - MIA
- /[NO]TIE (Alpha only)—Enables the compiled code to be used in combination with translated images, either because the code might call into a translated image or might be called from a translated image.
- /[NO]UNSIGNED_CHAR—By default, `char` is a signed character type. The /UNSIGNED_CHAR qualifier lets you change this default to an unsigned character type, which causes all plain `char` declarations to have the same representation and set of values as `unsigned char` declarations. The default is /NOUNSIGNED_CHAR.
- /VAXC (VAX only)—Invokes the VAX C compiler. The default is set to either /DECC or /VAXC during installation.

A.1.2. Comment Processing

VAX C treats a comment in a macro definition as if the comment were replaced with no characters. This allows comments to paste tokens together, as in the following example:

```
#define PASTE(X) X/*  */1
int PASTE(VAR);
```

This example declares the variable `VAR1`. Standard C requires that comments be treated as if they were replaced by a single space. In VSI C, therefore, comments cannot be used to concatenate tokens when /STANDARD=ANSI89 or /STANDARD=RELAXED is specified. (The new operator `##` is provided to allow token concatenation in macros).

VSI C for OpenVMS systems continues to replace comments with no characters when /STANDARD=VAXC or /STANDARD=COMMON is specified; and /WARN=ENABLE=CHECK provides a diagnostic to flag comments that are used to concatenate tokens.

For /STANDARD=COMMON and /STANDARD=RELAXED, C++ style comments (//) are supported.

A.1.3. String Literal Concatenation

VSI C introduces a new ANSI-conforming feature that allows convenient continuation of string literals. If string literals are separated only by white space, the string literals are concatenated to form one string literal. For example:

```
fputs("This is really "
```

```
   "one string literal", stderr);
```

String literal concatenation works for both normal string literals and wide string literals.

A.1.4. Recursive main() Function

In VAX C, `main`, or any function using the VAX C `main_program` option, is not recursively reentrant.

As required by standard C, the main function in VSI C can now be called recursively.

A.1.5. Trigraph Sequences

Standard C defines an additional representation of some of the special characters in the C language source abstract character set. These additional representations are sequences of three characters called *trigraphs*. *Table A.1, "Trigraphs"* lists the trigraphs and the character each is mapped to.

Table A.1. Trigraphs

Trigraph	Replacement
??=	#
??(	[
??/	\
??)	]
??'	^
??<	{
??!	|
??>	}
??-	~

Conceptually, every trigraph is removed from the file and its replacement is substituted. Each ? that does not begin one of the trigraphs is not changed. Trigraph processing occurs before tokenization takes place. Thus, even trigraphs in string constants have their replacements substituted.

Trigraph support has the potential to change the meaning of existing C code that unintentionally contains a trigraph in a string literal. However, since such conflicts will be quite rare, there is no facility for disabling trigraph support.

Trigraph support is available in strict and relaxed mode.

A.1.6. Alert Escape Sequence

As specified by standard C, VSI C defines a new escape sequence for the alert character. The escape sequence `\a` represents the ASCII BEL (Ctrl/G) character.

A.1.7. Hexadecimal Escape Sequence

VAX C limits hexadecimal escape sequences to at most 3 hex digits, but standard C allows an unlimited number of digits. VSI C removes the limit imposed by VAX C.

This can cause some programs to behave differently. The string `"\x0012"` is currently interpreted by VAX C as a string with two characters in it: a Ctrl/A followed by the character “2”. Under standard C rules, the string consists of a single character whose character code is hexadecimal 12 (Ctrl/R). However, this problem is unlikely to occur in practice.

A.1.8. Invalid Escape Sequences

VSI C issues a warning message if it encounters an invalid escape sequence. VAX C did not diagnose such usage.

A.1.9. $ in Macro Names

The dollar sign ($) is not an element of the minimum basic character set allowed by the C standard. By a systemwide convention, the dollar sign identifies VSI reserved identifiers. VSI C for OpenVMS systems supplies header files containing many macros with dollar signs in their names, and the VAX C compiler predefines some macros with dollar signs in their names. In strict ANSI C mode, such macros trigger a warning.

A.1.10. Null Arguments to Macros

In VSI C, null arguments to a macro produce a BUGCHECK. VAX C allowed macro arguments to be null.

A.1.11. Standard C Name Space Conformance

Standard C strictly controls the name space of C programs, and prohibits compilers or their standard-specified header files from intruding on the name space reserved for user programs. Specifically, the C Standard requires that compiler extensions begin with an underscore followed by an uppercase letter or another underscore.

This affects VAX C extensions involving additional keywords and predefined macros. It also affects the freedom of VSI C to add additional macros, variables, and functions to the standard-specified header files, such as `<stdio.h>`.

The following sections describe how VSI C solves the reserved name space problem for extensions involving keywords, predefined macros, and header file contents.

A.1.11.1. Nonstandard Keywords

VAX C has several keywords that intrude into the user name space. The VSI C compiler in strict ANSI C mode (/STANDARD=ANSI89) does not recognize keywords that are VAX C-specific extensions to the language. They are recognized instead as identifier names. As a result, programs that use these extensions as keywords cannot be compiled in strict ANSI C mode without eliciting syntax errors.

Similarly, the VSI C compiler in VAX C mode and relaxed ANSI C mode does recognize keywords that are VAX C-specific extensions to the language. Therefore, programs that use these names as identifiers cannot be compiled in VAX C or relaxed mode without eliciting syntax errors. In relaxed mode, the compiler generates a warning for these keywords. When the /STANDARD=ANSI89 qualifier is used, the compiler strictly follows the ANSI C rules about the name space, and does not recognize the old spellings as keywords.

Table A.2, "Nonstandard Keywords" shows the traditional spelling and the new spelling of the keywords affected, as well as their corresponding standard-conforming pragmas.

Table A.2. Nonstandard Keywords

Keyword	Corresponding Standard-Conforming Pragma
globaldef	#pragma extern_model
globalref	#pragma extern_model
globalvalue	#pragma extern_model
noshare	#pragma extern_model
readonly	#pragma extern_model

A.1.11.2. Nonstandard Predefined Macros

Alternate spellings that follow standard C rules are added to VSI C for all VAX C predefined macros. For compatibility, both the old spellings of the predefined macros and the new spellings are recognized by the compiler. However, when the /STANDARD=ANSI89 qualifier is used, the compiler strictly follows the C standard's rules about the name space, and does not recognize the old spellings as predefined macros. You are encouraged to use the new standard C conformant spelling of the macros.

Table A.3, "New and Traditional Spellings of Macros" shows the traditional spelling and the new spelling of the predefined macros affected

Table A.3. New and Traditional Spellings of Macros

Traditional Spelling	New Spelling
vax	__vax
vax11c	__vax11c
vaxc	__vaxc
VAX	__VAX
VAX11C	__VAX11C
VAXC	__VAXC
vms	__vms
VMS	__VMS
vms_version	__vms_version
VMS_VERSION	__VMS_VERSION

A.1.11.3. Nonstandard Identifiers in Standard-Specified Header Files

The C standard specifies exactly what identifiers in the normal name space are declared by the standard header files. A compiler is not free to declare additional identifiers in a header file unless the identifiers follow defined rules (the identifier must begin with an underscore followed by an uppercase letter or another underscore).

When running the VSI C compiler on OpenVMS systems in strict ANSI C mode (/STANDARD=ANSI89), versions of the standard header files are included that hide many identifiers that do not follow the rules. The <stdio.h> header file, for example, hides the definition of the macro

TRUE. The compiler accomplishes this by predefining the macro __HIDE_FORBIDDEN_NAMES in strict ANSI C mode.

You can use the command-line qualifier /UNDEFINE="__HIDE_FORBIDDEN_NAMES" to prevent the compiler from predefining this macro, thus including macro definitions of the forbidden names.

The header files are modified to only define additional VAX C names if __HIDE_FORBIDDEN_NAMES is undefined. For example, <stdio.h> might contain the following:

```
#ifndef __HIDE_FORBIDDEN_NAMES
#define TRUE    1
#endif
```

A.1.12. VSI C Predefined Macros

VSI C for OpenVMS Systems supports the following new system-identification macros:

```
__DECC
__alpha
__ALPHA
__Alpha_AXP
__32BITS
__mia
__STDC__
```

A.1.13. VSI C Types

The following sections describe changes to the data types supported by VSI C.

A.1.13.1. signed Reserved Word

VSI C supports the new reserved word signed to complement unsigned. The signed keyword may be used with the char, short, int, and long keywords to specify the types signed char, signed short, signed int, and signed long. (These types are already supported by VAX C.) The signed keyword can also be used when declaring bit fields to specify explicitly that the bit field is signed.

Standard C specifies that signed short, signed int, and signed long are the same types as short, int, and long, respectively. However, signed char is not the same type as char, even though VSI C uses the same representation for both of them. This does not affect normal mixing of the two types, but it does mean that in VSI C a pointer to signed char is not compatible with a pointer to char. Note that programs that previously used signed as an identifier will now be in error, even in VAX C mode. The /[NO]UNSIGNED_CHAR qualifier can be used to specify whether char is signed or unsigned.

A.1.13.2. Removal of the long float Type

In VAX C, long float is a synonym for double. Since the C Standard retires the long float specification, VSI C in strict ANSI C mode diagnoses any use of long float as an error. The long float type is still accepted as a synonym for double in VAX C mode, but it elicits a warning diagnostic to the effect that this is an obsolete usage.

A.1.13.3. Addition of the long double Type

On OpenVMS VAX systems, VSI C maps the standard C defined `long double` type to the G_floating or D_floating format, depending on the /FLOAT (or /[NO]G_FLOAT) qualifier used. (VAX only)

On OpenVMS Alpha systems, the `long double` type defaults to X_floating (/L_DOUBLE_SIZE=128). If /L_DOUBLE_SIZE=64 is specified, the `long double` type is mapped to G_floating, D_floating, or T-floating, depending on the /FLOAT (or /[NO]G_FLOAT) qualifier used. (Alpha only)

The `<float.h>` header file is modified to define the appropriate values to describe the characteristics of this new data type.

A.1.13.4. Addition of Processor-Specific Integer Data Types

VSI C for OpenVMS Systems supports the following processor-specific integer data types:

- `__int16`
- `__int32`
- `__int64` (Alpha only)

These data types are intended for applications that need integer data types of a specific size across platforms that support the data type.

The `<ints.h>` header file contains `typedef`s for the signed and unsigned variations of these integer data types. For increased portability, use these `typedef`s rather than using the built-in data types directly.

Note that the 64-bit integer types are available on OpenVMS Alpha systems but not on OpenVMS VAX systems.

The contents of `<ints.h>` are:

```
#ifndef __INTS_LOADED
#define __INTS_LOADED 1 /
*****************************************************************************
**
**  <ints.h> - Definitions for platform specific integer types
**
*****************************************************************************
**  Header is nonstandard
*****************************************************************************

#pragma __nostandard
 /*
**  Ensure that the compiler will not emit diagnostics about "signed"
**  keyword usage when in /STAND=VAXC mode (the reason for the diagnostics
**  is that VAX C does not support the signed keyword).
*/
#if ((__DECC_VER >= 50600000) && !defined(__DECCXX))
#   pragma __message __save
#   pragma __message __disable (__SIGNEDKNOWN)
    typedef   signed char  int8;
    typedef unsigned char uint8;
#   pragma __message __restore
```

```
#else
    typedef   signed char  int8;
    typedef unsigned char uint8;
#endif
 /*
**  Define 16 and 32 bit integer types
*/
#if defined(__DECC) || (defined(__DECCXX) && defined(__ALPHA))
    typedef          __int16 int16;
    typedef unsigned __int16 uint16;
    typedef          __int32 int32;
    typedef unsigned __int32 uint32;
#else
    typedef          short int int16;
    typedef unsigned short int uint16;
    typedef          int int32;
    typedef unsigned int uint32;
#endif
 /*
**  Define 64 bit integer types only on Alpha
*/
#ifdef __ALPHA
    typedef          __int64 int64;
    typedef unsigned __int64 uint64;
#endif

#pragma __standard
#endif /* __INTS_LOADED */
```

A.1.14. Type Compatibility

VSI C for OpenVMS systems in strict and relaxed mode uses different rules than VSI C in VAX C mode to determine if two types are identical:

- VAX C mode treats `int` and `long` as exactly the same type. ANSI C mode differentiates between `int` and `long` even if both types use the same underlying representation.
- VAX C mode treats `char` and `signed char` as exactly the same type. ANSI C mode differentiates between `char` and `signed char`, even though the same underlying representation is used for both types.
- VAX C mode treats two structure or union types as the same type if they have the same size in bytes. In ANSI C mode, a structure or union type is compatible only with itself.
- VAX C mode, by default, treats all pointer types as if they were compatible. ANSI C mode defines two pointer types as being compatible only if they are identically qualified pointers to compatible types.

These rules cause the strict and relaxed modes to be much more strict than VAX C mode about type checking.

A.1.15. Composite Types

As required by the C standard, VSI C merges type information from two declarations of the same object in the same scope. The declarations are required to be type-compatible and the linkage of the declarations must be such that multiple declarations in the same scope are allowed.

The composite type (the merged type) can be formed only from array or function types. Array types can have their array bounds specified, and function types can have their arguments specified.

For example, consider the following two declarations in the same scope:

```
extern int f(int (*)(), double (*)[3]);
extern int f(int (*)(char *), double (*)[]);
```

The resulting type for `f` is:

```
extern int f(int (*)(char *), double (*)[3]);
```

The VAX C compiler did not support composite types, although it might have appeared to do so. For example, in VAX C, what appears to be a second declaration of a composite function type, is actually a redeclaration of the function. This might have an effect on the compilation. For example, if the first declaration has ellipses and the second declaration does not, a composite type cannot be formed (not allowed by the C Standard). However, a redeclaration is done.

Since the composite type feature of the C standard is important even to those programming in VAX C mode, it is supported in VAX C mode. Therefore, it is possible to encounter declaration combinations that compile under VAX C but not under VSI C in VAX C mode.

A.1.16. Enumerations Have Type int

For type-checking purposes, VAX C previously considered enumeration types to be distinct from each other, and from the integer types, even though enumeration constants and variables have always been usable as ordinary integers. Since the VAX C model of enumerations was overly restrictive even from the strong typing point of view, and since such checking is not common in modern C, VSI C does not treat enumerations as a special type.

A.1.17. long double Constants

As specified by standard C, VSI C floating-point constants suffixed by `l` or `L` have type `long double`. (Currently, VAX C gives such constants type `double`).

A.1.18. Implicit Unsigned Integer Constants

A.1.18.1. OpenVMS VAX Systems

The type of an unsuffixed decimal integer constant is the first type in the following list that can represent its value: `int`, `long int`, or `unsigned long int`. (VAX only)

The type of an unsuffixed octal or hex constant is the first type in the following list that can represent its value: `int`, `unsigned int`, `long int`, or `unsigned long int`. (VAX only)

A.1.18.2. OpenVMS Alpha Systems

The type of an unsuffixed decimal integer constant is the first type in the following list that can represent its value: `int`, `long int`, `unsigned long int` (only in VAXC, COMMON, ANSI89, and MIA modes), `long long int`, `unsigned long int`. (Alpha only)

The type of an unsuffixed octal or hex constant is the first type in the following list that can represent its value: `int`, `unsigned int`, `long int`, `unsigned long int`, `long long int`, `unsigned long long int`. (Alpha only)

For more information, see the *Integer Constants* section in Chapter 1 of the *VSI C Reference Manual* [https://docs.vmssoftware.com/vsi-c-language-reference-manual/].

A.1.19. Multibyte and Wide Character Support

To meet the needs of non-European languages with large character sets, the C standard includes a framework to support characters encoded in multiple bytes. This framework is general enough to support character-processing extensions and character-set encodings already used in Asia, and allows for support for the draft proposed ISO Standard 10646, a multiple octet-coded character set that supports dozens of natural languages.

Standard C supports natural languages with large character sets by recognizing that normal character constants and string literals can be used to represent *multibyte characters*. A multibyte character is an encoding of variable-length characters where one, two, or more bytes in the string represents a single character in the natural language. The encoding is allowed to support locking shift states that change the encoding of characters for as long as the shift state holds.

Multibyte characters can occur in comments, character constants, and string literals.

Because string manipulation is very difficult when the character size varies from character to character, Standard C supports a fixed-size representation where each character is stored in the same number of bytes. This representation is called *wide character* support. VSI C supports a new form of wide character constant and wide string literal.

A.1.19.1. The Wide Character Type

Standard C requires that wide characters be represented by an integral type, and that there be a `typedef` named `wchar_t` for that type in the header `<stddef.h>`.

VSI C defines `wchar_t` to be `unsigned int`. This allows all character sets supported by ISO 10646 to be supported simultaneously.

A.1.19.2. Multibyte Characters in Comments, Character Constants, and String Literals

Full multibyte support requires that the compiler be able to determine whether an individual byte in a multibyte string is a single byte character or part of a multiple byte character. For example, the compiler must be able to distinguish between the single byte quote ending a string literal and a quote that is embedded in a multiple byte character and does not end the string literal.

A.1.19.3. Wide Character Constants

As required by standard C, VSI C supports wide character constants. The form of such a constant is the uppercase letter `L`, followed by a single quote, followed by a multibyte character, followed by a single quote.

The compiler collects the bytes making up the multibyte character into a string, and then calls the VSI C RTL `mbtowc` function to convert the multibyte character into a wide character. The resulting value has type `wchar_t`.

A.1.19.4. Wide String Literals

As required by the C Standard, VSI C supports wide string literals. The form of such a literal is the same as a normal string literal prefixed by the uppercase letter `L`.

The compiler collects the bytes making up the wide string literal into a string, and then calls the VSI C RTL `mbstowcs` function to convert the multibyte characters into wide characters. The resulting wide character string literal has type array of `wchar_t`.

A.1.20. Usual Arithmetic Conversions

In VSI C, the usual arithmetic conversions now support the `long double` type: if either operand of a binary operator that uses these conversions is `long double`, then the other operand is converted to `long double`.

A.1.21. Indexing as a Commutative Operator

As required by the C Standard, VSI C now defines the array indexing operator, [], as commutative. Thus, if `a` is an array and `i` is an integer, both `a[i]` and `i[a]` are valid.

A.1.22. Cast Operators

Standard C specifies that result of the cast operator is not an lvalue. However, VAX C does allow the cast operator to produce an lvalue.

The VSI C compiler in VAX C mode allows the cast operator to produce an lvalue.

A.1.23. Function Calls

The following sections describe changes to function calls.

A.1.23.1. Assignment Compatibility Argument Checking

Standard C defines a function call made with a prototype in scope as assigning the arguments to the parameters of the function. This means that all of the normal type checking and implied conversions that occur during an assignment take place when calling a function.

VAX C currently follows this model with two exceptions. First, it only performs the required type checking if /STANDARD=PORTABLE is given. Second, the assignment compatibility rules used by VAX C are not as stringent as the rules required by standard C. For example, two structs are assignment-compatible in VAX C only if they are the same size.

The VSI C compiler in VAX C mode and common mode is compatible with VAX C in assignment compatibility rules. Other modes follow the stricter standard C rules, documented in *Section A.1.27, "Assignment Compatibility"* of this guide, and issue the required messages even when /STANDARD=PORTABLE is not specified.

A.1.23.2. Passing Narrow Types to Old Syntax Functions

Traditionally, a function written in C was always called with widened argument types. (Arguments of narrow types like `char`, `short`, or `float` were passed as the widened types `int`, `int`, and `double`, respectively.) The C Standard preserves this calling mechanism for functions declared using the old syntax. Functions declared using the new prototype syntax may be called with narrow argument types.

Tradition, however, did not specify how the compiler was to interpret a function definition that declared formal arguments of narrow type. One interpretation was that the widened types actually passed should be converted to the narrow type of the formal declaration by the function in its prologue. Another

interpretation was that the compiler should rewrite the formal declarations to match the type of the argument actually passed. For example, under this second interpretation, the compiler would change a declaration of a formal argument of type `float` to a declaration of type `double`.

Standard C has standardized the first interpretation of a function with formal arguments of narrow types. VSI C for OpenVMS systems uses the standard C interpretation in all modes.

A.1.24. “Address of” Operator

In VSI C, if the argument of the unary `&` operator is an array, the result now has the type “pointer to array”. Previously, in VAX C, the result would have the type “pointer to the element type of the array”.

A.1.25. Unary Plus

VSI C supports the new standard C operator, unary plus (+). This operator returns the value of its operand (possibly widened by the integral promotions).

A.1.26. Relational Operators

As required by standard C, VSI C issues a warning (in all modes except VAX C mode) to diagnose a constraint violation if one of the operands of a relational operator is a pointer to a function. For example, the following code would issue a warning:

```
int (*f)();
if (f > NULL)
```

Note that it is valid to use the equality operators to compare function pointers.

A.1.27. Assignment Compatibility

Standard C has tighter assignment compatibility rules than those previously enforced by VAX C. (Note that assignment compatibility rules also control function argument passing.) VSI C assignment compatibility differs from that of VAX C in the following ways:

- An error is issued if a structure or union type is assigned to a different structure or union type, except in VAX C mode where it is allowed if the structure or union types have the same size.
- An error is issued if a non-`void` pointer type is assigned to a different non-`void` pointer type, except in VAX C mode where it is allowed.
- An error is issued if a `void` pointer type (except for a null constant pointer) is assigned to a pointer to a function (or vice versa), except in VAX C mode where it is allowed.

A.1.28. Declarations

Function prototype support, the new `const` and `volatile` type qualifiers, and the `void` type, were already implemented in VAX C. The following sections describe the additional VSI C support that affects declarations. References are to the relevant sections in the C Standard.

A.1.28.1. Implementation Limits

The C Standard requires that an implementation support certain minimum requirements; these are listed in the referenced section. In those cases where VAX C imposes a fixed limit, that limit has always met or exceeded the Standard's requirements, and programs that exceed any of these limits elicit the appropriate

errors. In strict ANSI C mode, VSI C now issues diagnostics against any source program constructs that exceed any of the Standard limits as well.

A.1.28.2. Identifier Name Length

In strict ANSI C mode, VSI C now issues diagnostic messages against declarations of external names in excess of six characters, or external names that are intended to denote different objects but that have the same spelling, and ignores alphabetical case.

A.1.28.3. Diagnosing Empty Declarations

The C Standard invalidates empty declarations, except for two special cases: one involving structure/union tags and the other involving the enumeration type. In strict ANSI C mode, VSI C issues an error message against any declaration that does not declare at least one of the following: a declarator, a tag, or the members of an enumeration.

A.1.28.4. Restriction on Placement of Storage-Class Specifiers

The C Standard specifies that allowing the placement of any storage-class specifier other than at the beginning of a declaration is an obsolete feature. In strict ANSI C mode, VSI C now issues an informational diagnostic to that effect when appropriate.

A.1.28.5. Diagnosing Old-Style Function Declarations

The C Standard specifies that old-style function declarations and definitions (that is, those not using the function prototype format) are obsolete. Old-style function declarations and definitions cause an informational message to be issued in all modes except VAX C.

A.1.28.6. Function Definitions Using typedef-names

The C Standard restricts the form of the declarator in a function definition: the function type itself may not be inherited from a `typedef`-name; that is, the declarator must explicitly contain a (possibly empty) parenthesized parameter list. If not, VSI C in strict ANSI C mode issues an error message.

A.1.28.7. Initialization

VSI C for OpenVMS systems supports the initialization of unions.

In VAX C, an aggregate initializer consisting of a single item does not have to have the outer braces. The outer braces are required by the C Standard.

VSI C allows this case in VAX C mode.

A.1.29. Bit-Field Initialization

The VSI C compiler initializes bit-field structure members differently than VAX C does. See *Section 4.7.2, "Bit-Field Initialization"*.

A.1.30. The Preprocessor

The following sections describe the differences between the VAX C and the VSI C preprocessors. Most of these differences reflect the VSI C preprocessor's conformance to the C Standard. References are to the relevant sections in the C Standard.

Note that most VAX C-specific preprocessor extensions are unaffected by these changes. These extensions continue to be supported quietly in VAX C mode, but elicit appropriate diagnostics in strict ANSI C mode.

A.1.30.1. White Space Appearing Before the

The C Standard removes the VAX C restriction that requires the # character introducing a preprocessor directive to always appear in column 1 of the source line. In VSI C, white space and comments can now precede the # on the same line.

A.1.30.2. The #define Directive and Macro Substitution

Before the C Standard, the lack of a precise definition of the behavior of macro expansion led to a number of inconsistencies among different C implementations. VSI C, in adhering to the C Standard, removes these and many other discrepancies by specifying precisely how macro substitution is to be performed:

- Except when running in VAX C or common mode, macro arguments are not allowed to replace parameters appearing within character strings in the macro definition.
- Except when running in VAX C or common mode, tokens within a macro definition are not concatenated if they were separated only by a comment; embedded comments are replaced by a blank.
- In all modes, keywords are allowed to be defined as macros.
- Macros are not replaced recursively in any mode except VAX C mode.

As required by the C Standard, VSI C supports two new operators that can appear only within macro definitions:

- The # operator takes a macro parameter as its operand and creates a character string from it. Combined with the new rule that adjacent character strings are implicitly concatenated into a single string, this provides the same capability as allowing substitution within strings.
- The ## operator concatenates the tokens on either side of it into a single token.

The C Standard also makes specific the sequence in which rescanning and further substitution is to take place, and under what conditions substitution does not take place. The C Standard also specifies under what circumstances a macro may be redefined: only benign redefinition is allowed, permitting a macro to be redefined only if the new definition is token-wise identical to the old definition.

A.1.30.3. The #line Directive

The C Standard specifies that macro substitution can occur on the operands of the `#line` directive, that the line number operand is restricted to the range 1 to 32,767, and that the file name operand must be treated as any character string literal. VAX C did not support macro substitution on this directive, performed no range checking on the line number, and restricted the length of the character string to 255.

VSI C supports macro substitution on the `#line` directive, diagnoses an out-of-range line number (in strict ANSI C mode only), and allows the file name character string to be as long as the maximum length supported by the compiler for ordinary strings. (Note that the C Standard requires support for a minimum of 509 characters in a string, and that VSI C supports strings up to 65,535 characters.)

A.1.30.4. The #error Directive

VSI C in both strict ANSI C mode and VAX C mode supports the new `#error` directive required by the C Standard.

A.1.30.5. The #pragma builtins Directive

The `#pragma builtins` directive is provided for VAX C compatibility.

VSI C implements `#pragma builtins` by including the `<builtins.h>` header file, and is equivalent to `#include <builtins.h>` on OpenVMS systems.

This header file contains prototype declarations for the built-in functions that allow them to be used properly. By contrast, VAX C implemented this pragma with special-case code within the compiler, which also supported a `#pragma nobuiltins` preprocessor directive to turn off the special processing. Because declarations cannot be "undeclared," VSI C does not support `#pragma nobuiltins`. Furthermore, the names of all the built-in functions use a naming convention defined by standard C to be in a namespace reserved to the C language implementation.

A.1.30.6. The #pragma dictionary Directive

The `#pragma dictionary` preprocessor directive replaces the `#dictionary` directive, but the latter is still supported in VAX C mode for compatibility.

The `#pragma dictionary` and `#dictionary` preprocessor directives now allow you to specify whether all string data type variables should be null-terminated.

A.1.30.7. The #pragma extern_model Directive

The `#pragma extern_model` directive is added to control the compiler's interpretation of objects that have external linkage. This pragma lets you choose the global symbol model to be used for external variables.

A.1.30.8. The #pragma linkage Directive (Alpha only)

The `#pragma linkage` preprocessor directive allows you to specify special linkage types for function calls.

A.1.30.9. The #pragma use_linkage Directive (Alpha only)

The `#pragma use_linkage` directive associates a previously defined special linkage with a function.

A.1.30.10. The #pragma message Directive

The `#pragma message` directive controls the issuance of individual diagnostic messages or groups of messages. Use of this pragma overrides any command-line options that may affect the issuance of messages.

A.1.30.11. The #pragma module Directive

The `#pragma module` preprocessor directive replaces the `#module` directive, but the latter is still supported in VAX C mode for compatibility.

A.2. Features Affecting the VSI C Run-Time Library and Include Files

This section describes new features pertaining to the standard header files in the VSI C Run-Time Library (C RTL).

A.2.1. <stddef.h>

The wchar_t type is now added to this header file. The declaration of errno is also removed.

A.2.2. <ctype.h>

Because the C Standard refers to the macros in <ctype.h> as functions, the <ctype.h> header file now includes function prototypes for functions in the VSI C RTL that perform the same operations as the macros currently defined in this header file. These functions have been added to the VSI C RTL.

The nonstandard toascii macro remains because, according to the C Standard, Section 4.14.2, names beginning with “to” are reserved by the C Standard when <ctype.h> is included.

A.2.3. <fp_class.h>

This header file containing IEEE floating-point class constants has been added to support the new VSI C RTL functions fp_class, fp_classf, and fp_classl available on OpenVMS Alpha systems.

A.2.4. <locale.h>

The new standard header file <locale.h> is now supported and includes prototypes for the functions setlocale and localeconv, which have been added to the VSI C RTL.

A.2.5. <math.h>

The functions cabs and hypot are no longer defined in the <math.h> header file when the compiler is run in strict ANSI C mode.

A.2.6. <signal.h>

The SIGABRT signal is implemented and defined in the <signal.h> header file. SIG_ATOMIC_T is now defined as char. In strict ANSI C mode, the following are not declared: ssignal, gsignal, kill, pause, sleep, sigvec, sigblock, sigsetmask, sigstack, and sigpause.

In strict ANSI C mode, the names of the ILL_* and FPE_* macros are changed to begin with “SIG” (for example, SIGILL_RESAD_FAULT, SIGFPE_INTOVF_TRAP, and so on) or be removed.

The BADSIG macro is renamed to SIG_ERR.

A.2.7. <stdio.h>

The <stdio.h> header file now defines the type size_t and no longer includes <stdarg.h>. The v*printf functions are now prototyped using the type that va_list is defined to be (that is, char *).

In strict ANSI C mode, the following macros are not visible: TRUE, FALSE, SEEK_EOF, OPEN_MAX, L_ctermid, L_cuserid, L_lcltmpnam, L_nettmpnam, and FILE_TYPE. In strict ANSI C mode, the following functions are not visible: fgetname, fdopen, getw, and putw.

The rename function is added.

The fflush function is modified so that a null argument causes it to flush all files.

The printf function is modified to provide the following support:

- The %i conversion is supported.
- The "0" flag works properly in conjunction with other flags and with all conversion specifiers. The long double type and the h modifier are now supported.
- The %d and %i specifiers interpret the precision specification.

The scanf function is modified to handle white space as specified by the C Standard. The %p specifier is added. The L flag for long double is added.

The clearerr, feof, and ferror macros are now provided as both macros and functions. By default, they are accessed as macros. To access them as functions, perform an #undef on the macro of the same name. For example:

```
#undef clearerr
```

A.2.8. <stdlib.h>

The <stdlib.h> header file is modified to define size_t and wchar_t directly, rather than including <stddef.h>. The names of the DIV_T and LDIV_T structures now begin with underscores.

The MB_CUR_MAX macro is added.

The multibyte character and string functions mblen, mbtowc, wctomb, mbstowcs and wcstombs are added as specified in Sections 4.10.7 and 4.10.8 of the C Standard.

The abort() function is changed to only raise a SIGABRT signal.

A.2.9. <string.h>

The strcoll and strxfrm functions are added as specified in the C Standard, Sections 4.11.4.3 and 4.11.4.5.

A.2.10. <time.h>

In strict and relaxed modes, the following changes apply to the <time.h> header file:

- CLOCKS_PER_SEC is defined instead of CLK_TCK.
- The size_t and time_t types are defined, and the <types.h> and <timeb.h> header files are not included.
- The times and ftime functions, and the struct tbuffer, tbuffer_t, and tm_t types must not be defined in your source code.

The `mktime` and `strftime` functions are added as specified in the C Standard, Sections 4.12.2.3 and 4.12.3.5, respectively.

A.3. Unsupported Features

VSI C for OpenVMS systems does not support parallel processing on either OpenVMS VAX or OpenVMS Alpha systems and, therefore, does not support the following qualifiers and preprocessor directives:

/[NO]PARALLEL
/SHOW=NODECOMPOSITION

```
#pragma ignore_dependency
#pragma safe_call
#pragma sequential_loop
```

Appendix B. Common Pitfalls

This appendix contains some of the most common pitfalls you might encounter while using VSI C. Symptoms, examples, and solutions are described.

Symptom:

The compiler generates an "Insufficient Virtual Memory" error.

Solution:

Increase the PAGEFILEQUO process quota and/or the VIRTUALPAGCNT sysgen parameter.

Symptom:

The compiler does not recognize expected routine entry points.

Example:

```
$ type main.c
main()
{
  exit(1);
}
$ cc main.c
exit(1);
..^
%CC-I-IMPLICITFUNC, In this statement, the identifier
exit is implicitly declared as a function.
```

Solutions:

1. In ANSI mode, include function prototypes (such as `#include <stdlib.h>`) in this example.

2. Compile using the /STANDARD=VAXC qualifier.

Symptom:

The compiler generates a %CC-E-NOTCOMPAT error message for seemingly correct code.

Example:

```
$ type main.c
void foo(short a);
void foo(a)
  short a;
{}
$ cc main.c
```

```
void foo(a)
.....^

%CC-E-NOTCOMPAT, In this declaration, the type of foo is not compatible
with the types of previous declarations of foo.
```

This example represents a mixing of new-style function prototypes and old-style function declarations. In the following declaration, the argument `a` gets widened to `int` on entry to `foo` before being converted to type `short`:

```
void foo(a)
short a;
```

Consequently the compiler detects a type mismatch. The example can be generalized to `float` variables, or any combination of (`unsigned`) `char` or `short` arguments.

Solutions:

1. Replace the new-style function prototype with an old-style function definition:

   ```
   void foo();
   void foo(a)
   short a;
   {}
   ```

2. Replace the old-style function declaration with a new-style function declaration:

   ```
   void foo(short a);
   void foo(short a)
   {}
   ```

Symptom:

Include-file lookups do not include the anticipated files.

Example:

By default, VSI C for OpenVMS Systems first searches the directory containing the top-level source file. Consider the following files and the `#include` statements they contain:

```
[]main.c
   #include "[.sub1]a.h"

[.sub1]a.h
   #include "b.h"

[.sub1]b.h
   "In [.sub1]"

[.sub2]b.h
  "In [.sub2]"
```

Compiling with the following command includes the `[.sub2]b.h` header file:

```
cc/include=[.sub2]main.c
```

Solution:

Specify /NESTED_INCLUDE_DIRECTORY in order to first search the directory containing the top-level source file (not the directory of the source file containing the `#include` directive).

Symptom:

VAX C extensions to the language are not accepted by the compiler.

Example:

```
int _align (word) w1;
....^
%CC-W-ALIGNEXT, _align is a language extension.
```

Solution:

Compile using the /STANDARD=VAXC qualifier.

Symptom:

The compiler generates a ADDRCONSTEXT (warning in /STANDARD=RELAXED mode and error in /STANDARD=ANSI mode) for seemingly correct code.

Example:

```
$ type main.c
struct dsc$descriptor_s
{
  unsigned short dsc$w_length;
  unsigned char dsc$b_dtype;
  unsigned char dsc$b_class;
  char *dsc$a_pointer;
};

main()
{
  char name[5];
  struct dsc$descriptor_s name_dsc = {
  sizeof(name)-1, 14, 1, name };
}

$ cc main.c

sizeof(name)-1, DSC$K_DTYPE_T, DSC$K_CLASS_S, name };
.........................................^
%CC-W-ADDRCONSTEXT, In the initializer for name_dsc.dsc$a_pointer,
"name" does not have a constant address, but occurs in a context that
requires an address constant. This is an extension of the language.
```

Solution:

The C Standard restricts allowable automatic aggregate initializion. The VSI C compiler does not have this restriction in /STANDARD=VAXC mode. Use any of the following solutions.

- Declare the array `name` to be static:

```
static char name[5];
```

- Compile in /STANDARD=VAXC mode.
- Compile with /WARNING=DISABLE=ADDRCONSTEXT.
- Insert the `#pragma [no]standard` preprocessor directive to suppress the warning message:

```
#pragma __nostandard
struct dsc$descriptor_s name_dsc = {
    sizeof(name)-1, DSC$K_DTYPE_T, DSC$K_CLASS_S, name };
     }
#pragma __standard
```

Appendix C. Programming Tools

This appendix provides information on tools that you can use to develop and to refine your VSI C programs. Some of the products described ship with the OpenVMS operating system; others must be purchased separately. The following products are described in this appendix:

- OpenVMS Debugger (*Section C.1, "OpenVMS Debugger"*)
- OpenVMS Text Processing Utility (*Section C.2, "OpenVMS Text Processing Utility "*)
- Language-Sensitive Editor and Source Code Analyzer (*Section C.3, "Language-Sensitive Editor and the Source Code Analyzer"*)
- CDD/Repository (*Section C.4, "CDD/Repository"*)

C.1. OpenVMS Debugger

A debugger is a tool to help you locate run-time errors quickly. It enables you to observe and manipulate the program's execution interactively, step by step, until you locate the point at which the program stopped working correctly.

The OpenVMS Debugger (provided with the OpenVMS operating system) is a symbolic debugger. You can refer to program locations by the symbols (names) you used for those locations in your program: the names of variables, routines, labels, and so on. You do not have to use virtual addresses to refer to memory locations.

If your program is written in more than one language, you can change from one language to another in the course of a debugging session. The current source language determines the format used for entering and displaying data, as well as other features that have language-specific settings (for example, comment characters, operators and operator precedence, and case sensitivity or insensitivity).

For information on the debugger, see the *VSI OpenVMS Debugger Manual*.

The following sections provide language-specific information on the OpenVMS Debugger.

C.1.1. Compiling and Linking to Prepare for Debugging

The following example shows how to compile and link a VSI C program (consisting of a single compilation unit named INVENTORY) so that you will be able to use the debugger:

```
$ CC/DEBUG/NOOPTIMIZE INVENTORY
$ LINK/DEBUG INVENTORY
```

The /DEBUG qualifier on the CC command causes the compiler to write the debug symbol records associated with INVENTORY.C into the object module, INVENTORY.OBJ. These records allow you to use the names of variables and other symbols declared in INVENTORY with debugger commands. (If your program has several compilation units, you must compile each unit that you want to debug with the /DEBUG qualifier.)

You should use the /NOOPTIMIZE qualifier when you compile in preparation for debugging. Without this qualifier, the resulting object code is optimized, which may cause the contents of some program locations to be inconsistent with what you might expect from the source code. (After the program has been debugged, you will probably want to recompile it without the /NOOPTIMIZE qualifier, because optimization might reduce a program's size and increase the execution speed.)

The /DEBUG qualifier on the LINK command causes the linker to include all symbol information that is contained in INVENTORY.OBJ in the executable image. The qualifier also causes the OpenVMS image activator to start the debugger at run time. (If your program has several object modules, you might need to specify other modules in the LINK command.)

C.1.2. Starting and Terminating a Debugging Session

Before you invoke the debugger, enter the following command to check the current debugger configuration:

```
$ SHOW LOGICAL DBG$PROCESS
%SHOW-S-NOTRAN, no translation for logical name DBG$PROCESS
```

If DBG$PROCESS has a value other than undefined (as in the previous example) or DEFAULT, enter the following command to change this value:

```
$ DEFINE DBG$PROCESS DEFAULT
```

Enter the DCL command RUN to invoke the debugger. The following message appears on your screen:

```
$ RUN INVENTORY

%DEBUG-I-INITIAL, language is C, module set to 'INVENTORY'
DBG>
```

You can now enter debugger commands at the DBG> prompt. At this point, if you enter the GO command, program execution begins and continues until it is forced to pause or stop (for example, if the program prompts you for input, or an error occurs).

To interrupt a debugging session and return to the debugger prompt, press Ctrl/C. This is useful if, for example, your program loops or you want to interrupt a debugger command that is still in progress. For example:

```
DBG> GO
.
.
.
(infinite loop)
Ctrl/C
Interrupt
%DEBUG-W-ABORTED, command aborted by user request
DBG>
```

The following message indicates that your program has completed successfully:

```
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG>
```

To end a debugging session, enter the EXIT command at the DBG> prompt or press Ctrl/Z:

```
DBG> EXIT
$
```

C.1.3. Notes on VSI C Support

In general, the OpenVMS Debugger supports the data types and operators of VSI C and of the other debugger-supported languages. To get information on the supported data types and operators of any of the languages, enter the HELP LANGUAGE command at the DBG> prompt.

The following sections present VSI C specific debugging examples.

C.1.3.1. Debugger Command-Line Options

VSI C provides a set of debugger options that you can specify to the /DEBUG qualifier to the CC command. These options alter the types of information that the compiler places in the object module for use by the OpenVMS Debugger. The debugger options include using traceback records, using the symbol table, and enabling the debugger to step into inline functions. For information about these options, see *Section 1.3.4, "CC Command Qualifiers"*.

C.1.3.2. Accessing Scalar Variables

The EXAMINE command displays scalar variables of any VSI C data type. Reference scalar variables in the same case that you declare them, using the VSI C syntax for such references.

Example C.1, "Debugging Sample Program SCALARS.C" shows the VSI C program SCALARS.C used in the examples that follow.

Example C.1. Debugging Sample Program SCALARS.C

```
/*  SCALARS.C  This program defines a large number of          *
 *             variables to demonstrate the effect             *
 *             of the various STEP debugger commands.          */

main()
{
   static float light_speed;       /*  Define the variable.    */
   static double speed_power;
   static unsigned ui;
   static long li;
   static char ch;
   static enum primary { red, yellow, blue } color;
   static long *ptr;

   light_speed = 3.0e10;
   speed_power = 3.12345678901234567890e10;
   ui = -438394;
   li = 790374270;
   ch = 'A';
   color = blue;
   ptr = &li;
}
```

The following debugging examples are based on executing SCALARS.EXE and show the commands used to access variables of scalar data type.

The debugger command SHOW SYMBOL/TYPE displays the data type of one variable:

```
DBG> show symbol/type color
data SCALARS\main\color
    enumeration type (primary, 3 elements), size: 4 bytes
```

The following debugging commands set a breakpoint before the end of the program and execute the program up to the breakpoint. The program initializes the variables declared in `main`:

```
DBG> set break %line 22
DBG> go
```

```
break at SCALARS\main\%LINE 22
    22: }
```

The EXAMINE command displays the contents of the variables listed. The `char` variables are interpreted by the debugger as byte integers, not ASCII characters:

```
DBG> examine li, ui, light_speed, speed_power, ch, color, *ptr
SCALARS\main\li:        790374270
SCALARS\main\ui:        4294528902
SCALARS\main\light_speed:       3.0000001E+10
SCALARS\main\speed_power:       31234567890.12346
SCALARS\main\ch:        65
SCALARS\main\color:     blue
*SCALARS\main\ptr:       790374270
```

To display the contents of `ch` as a character, you must use the /ASCII qualifier:

```
DBG> examine/ascii ch
SCALARS\main\ch:      "A"
```

The DEPOSIT command loads the value single_quote>z' in the variable `ch`; the EXAMINE command shows that single_quote>z' has replaced the previous contents of the variable `ch`. Again, use the /ASCII qualifier to translate the byte integer into its ASCII equivalent:

```
DBG> deposit/ascii ch = 'z'
DBG> examine/ascii ch
SCALARS\main\ch:        "z"
DBG>
```

C.1.3.3. Accessing Arrays

With the EXAMINE command, you can look at the values in arrays using VSI C syntax for array references. You can examine an entire array by giving the array identifier. You can examine individual elements of the array using the array operator ([ ]). Array elements can have any data type.

Consider the following declaration: `int arr[10];`

This declares an array of 10 elements, `arr[0]` through `arr[9]`.

Example C.2, "Debugging Sample Program ARRAY.C" shows the VSI C program ARRAY.C used in the examples that follow.

Example C.2. Debugging Sample Program ARRAY.C

```
/*  ARRAY.C  This program increments an array to              *
 *           demonstrate the access of arrays in VSI C.      */

main()
{
    int i;
    static int arr[10];
    for (i=0; i<10; i++)
          arr[i]=i;
}
```

The examples that follow are based on executing ARRAY.EXE and show the commands used to access variable arrays. (Note: Compile ARRAY.C with the /NOOPT qualifier for the examples to work as described).

The following commands set a breakpoint at the last line in the program and execute the program to that point:

```
DBG> set br  %line 10
DBG> go
break at ARRAY\main\%LINE 10
    10: }
```

By specifying the variable identifier, you can look at the entire array:

```
DBG> examine arr
ARRAY\main\arr
    [0]:        0
    [1]:        1
    [2]:        2
    [3]:        3
    [4]:        4
    [5]:        5
    [6]:        6
    [7]:        7
    [8]:        8
    [9]:        9
```

You can examine individual elements of the array by using the bracket operator to specify the subscript of the element. Pressing Return (the debugger's address reference operator) in an EXAMINE command displays the next element of the array. Using the up-arrow address reference operator (^) displays the previous member of the array:

```
DBG> examine arr[5]
ARRAY\main\arr[5]:      5
DBG> examine
ARRAY\main\arr[6]:      6
DBG> examine ^
ARRAY\main\arr[5]:      5
```

C.1.3.4. Accessing Character Strings

Character strings are implemented in VSI C as null-terminated ASCII strings (ASCIZ strings). To examine and deposit data in an entire string, use the /ASCIZ qualifier (abbreviated /AZ) so that the debugger can interpret the end of the string properly. You can examine and deposit individual characters in the string using the C array subscripting operators ([ ]). When you examine and deposit individual characters, use the /ASCII qualifier.

Example C.3, "Debugging Sample Program STRING.C" shows the VSI C program STRING.C used in the examples that follow.

Example C.3. Debugging Sample Program STRING.C

```
/*  STRING.C  This program establishes a string to             *
 *            demonstrate the access of strings in VSI C.      */

main()
{
   static char *s = "vaxie";
   static char **t = &s;
}
```

The following examples are based on executing STRING.EXE and show the commands used to manipulate C strings.

The EXAMINE/AZ command displays the contents of the character string pointed to by `*s` and `**t`:

```
DBG> step
stepped to STRING\main\%LINE 8
    8: }
DBG> examine/az  *s
*STRING\main\s: "vaxie"
DBG> examine/az  **t
**STRING\main\t:        "vaxie"
```

The DEPOSIT/AZ command deposits a new ASCIZ string in the variable pointed to by `*s`. The EXAMINE/AZ command displays the new contents of the string:

```
DBG> deposit/az  *s = "VSI C"
DBG> examine/az  *s,  **t
*STRING\main\s: "VSI C"
**STRING\main\t:         "VSI C"
```

You can use array subscripting to examine individual characters in the string and deposit new ASCII values at specific locations within the string. When accessing individual members of a string, use the /ASCII qualifier. A subsequent EXAMINE/AZ command shows the entire string containing the deposited value:

```
 examine/ascii s[2]
STRING\main\s[2]:    ' '
DBG> deposit/ascii s[2] = "-"
DBG> examine/az *s, **t
*STRING\main\s:    "VSI-C"
**STRING\main\t:   "VSI-C"
```

C.1.3.5. Accessing Structures and Unions

You can examine structures in their entirety or on a member-by-member basis, and deposit data into structures one member at a time.

To reference members of a structure or union, use the usual C syntax for such references. That is, if variable `p` is a pointer to a structure, you can reference member `y` of that structure with the expression `p ->y`. If variable `x` refers to the base of the storage allocated for a structure, you can refer to a member of that structure with the `x.y` expression.

The debugger uses the VSI C type-checking rules that follow to reference members of a structure or union. For example, in the case of `x.y`, `y` need not be a member of `x`; it is treated as an offset with a type. When such a reference is ambiguous—when there is more than one structure with a member `y`—the debugger attempts to resolve the reference according to the rules that follow. The same rules for resolving the ambiguity of a reference to a member of a structure or union apply to both `x.y` and `p ->y`.

- If only one of the members, `y`, belongs in the structure or union, `x`, that is the one that is referenced.
- If only one of the members, `y`, is in the same scope as `x`, then that is the one that is referenced.

You can always give a path name with the reference to `x` to narrow the scope that is used and to resolve the ambiguity. The same path name is used to look up both `x` and `y`.

Example C.4, "Debugging Sample Program STRUCT.C" shows the VSI C program STRUCT.C used in the examples that follow.

Example C.4. Debugging Sample Program STRUCT.C

```
/*  STRUCT.C  This program defines a structure and union       *
 *            to demonstrate the access of structures and     *
 *            unions in VSI C.                                */

main()
{
   static struct
      {
         int im;
         float fm;
         char cm;
         unsigned bf : 3;
      }  sv,  *p;

   union
      {
         int im;
         float fm;
         char cm;
      }  uv;

   sv.im = -24;
   sv.fm = 3.0e10;
   sv.cm = 'a';
   sv.bf = 7;          /* Binary: 111 */

   p = &sv;

   uv.im = -24;
   uv.fm = 3.0e10;
   uv.cm = 'a';

}
```

The following examples are based on executing STRUCT.EXE and show the commands used to access structures and unions.

The SHOW SYMBOL command shows the variables contained in the user-defined function main:

```
DBG> show symbol  *  in main
routine STRUCT\main
type STRUCT\main\char
data STRUCT\main\__func__
record component STRUCT\main\<generated_name_0002>.im
record component STRUCT\main\<generated_name_0002>.fm
record component STRUCT\main\<generated_name_0002>.cm
record component STRUCT\main\<generated_name_0002>.cm
data STRUCT\main\sv
data STRUCT\main\p
record component STRUCT\main\<generated_name_0001>.im
record component STRUCT\main\<generated_name_0001>.fm
record component STRUCT\main\<generated_name_0001>.cm
data STRUCT\main\uv
```

Set a breakpoint at line 29 and enter a GO command to initialize the variables declared in the structure sv:

```
DBG> set break  %line 29
DBG> go
break at STRUCT\main\%LINE 29
    29:    uv.im = -24;
```

Use the EXAMINE command with the name of the structure to display all structure members. Note that sv.cm has the char data type, which is interpreted by the debugger as a byte integer. The debugger also displays the value of bit fields in decimal:

```
DBG> examine sv
STRUCT\main\sv
    im: -24
    fm: .3000000E+11
    cm: 97
    bf: 7
```

To display the ASCII representation of a char data type, use the /ASCII qualifier on the EXAMINE command. To display bit fields in their binary representation, use the /BINARY qualifier:

```
DBG> examine/ascii sv.cm
STRUCT\main\sv.cm:     "a"
DBG> examine/binary sv.bf
STRUCT\main\sv.bf:     111
```

You deposit data into a structure one member at a time. To deposit data into a member of type char, use the /ASCII qualifier and enclose the character in either single or double quotation marks. To deposit a new binary value in a bit field, use the %BIN keyword:

```
DBG> deposit sv.im = 99
DBG> deposit sv.fm = 3.14
DBG> deposit/ascii sv.cm = 'z'
DBG> deposit sv.bf = %BIN 010
DBG> examine sv
STRUCT\main\sv
    im: 99
    fm: 3.140000
    cm: 122
    bf: 2
```

You can also access members of structures (and unions) by pointer, as shown in *p and p ->bf:

```
DBG> examine *p
*STRUCT\main\p
    im: 99
    fm: 3.140000
    cm: 122
    bf: 2
DBG> examine/binary p ->bf
STRUCT\main\p ->bf:     010
```

A union contains only one member at a time, so the value for uv.im is the only valid value returned by the EXAMINE command; the other values are meaningless:

```
DBG> step
```

```
stepped to STRUCT\main\%LINE 30
    30:    uv.fm = 3.0e10;
DBG> examine uv
STRUCT\main\uv
[Displaying union member number 1]
    im: -24
    fm: -1.5485505E+38
    cm: -24
```

This series of STEP and EXAMINE commands shows the content of the union as the different members are assigned values:

```
DBG> step
stepped to STRUCT\main\%LINE 31
    31:    uv.cm = 'a';
DBG> examine uv.fm
STRUCT\main\uv.fm:     .3000000E+11
DBG> step
stepped to STRUCT\main\%LINE 32
    33: }
DBG> examine/ascii uv.cm
STRUCT\main\uv.cm:     "a"
```

Example C.5, "Debugging Sample Program ARSTRUCT.C" shows the VSI C program ARSTRUCT.C used in the examples that follow.

Example C.5. Debugging Sample Program ARSTRUCT.C

```
/*  ARSTRUCT.C  This program contains a structure definition   *
 *              and a for loop to demonstrate the debugger's   *
 *              support for VSI C operators.          */

main()
{
   int   count,  i = 1;
   char  c = 'A';

   struct
      {
         int digit;
         char alpha;
      }  tbl[27], *p;

   for (count = 0; count <= 26; count++)
      {
         tbl[count].digit = i++;
         tbl[count].alpha = c++;
      }
}
```

The following examples are based on executing ARSTRUCT.EXE and show the use of C expressions on the debugger command line. (Note: Compile ARSTRUCT.C with the /NOOPT qualifier for the examples to work as described).

Relational operators can be used in expressions (such as `count == 2`) in a WHEN clause to set a conditional breakpoint:

```
DBG> set  break  %line  20  when  (count == 2)
```

```
DBG> go
break at ARSTRUCT\main\%LINE 20
    20:       }
```

The first EVALUATE command that follows uses C syntax to refer to the address of a variable. It is equivalent to the second command, which uses the /ADDRESS qualifier to obtain the address of the variable. The addresses of these variables might not be the same every time you execute the program if you relink the program.

```
DBG> evaluate  &tbl
2146736881
DBG> evaluate/address  tbl
2146736881
```

Individual members of an aggregate can be evaluated; the debugger returns the value of the member:

```
DBG> evaluate tbl[2].digit
3
```

When you perform pointer arithmetic, the debugger displays a message indicating the scale factor that has been applied. It then returns the address resulting from the arithmetic operation. A subsequent EXAMINE command at that address returns the value of the variable:

```
DBG> evaluate tbl + 4
%DEBUG-I-SCALEADD, pointer addition: scale factor of 5 applied to
right argument
2146736901
DBG> examine 2146736901
ARSTRUCT\main\tbl[4].digit:     5
```

The EVALUATE command can perform arithmetic operations on program variables:

```
DBG> evaluate tbl[4].digit * 2
10
```

The EVALUATE command can also perform arithmetic calculations that may or may not be related to your program. In effect, this command can be used as a calculator that uses C syntax for arithmetic expressions:

```
DBG> evaluate 7 % 3
1
```

The debugger enters a message when you use an unsupported operator:

```
DBG> evaluate count++
%DEBUG-W-SIDEFFECT, operators with side effects not supported (++, -)
```

C.1.3.6. Sample Debugging Session

Example C.6, "Debugging Sample Program POWER.C" shows the VSI C program POWER.C to be used in the sample debugging session shown in *Example C.7, "A Sample Debugging Session"*.

Example C.6. Debugging Sample Program POWER.C

```
/*  POWER.C  This program contains two functions: "main" and    *
 *           "power."  The main function passes a number to     *
 *           "power", which returns that number raised to the   *
 *           second power.                                      */
```

```
main()
{
    static int i, j;
    int power(int);

    i = 2;
    j = power(i);
}
power(int j)

{
    return (j * j);
}
```

Although this program contains no errors, *Example C.7, "A Sample Debugging Session"* shows some simple debugger commands that can be used to evaluate its execution. The callout numbers in this sample debugging session are keyed to the notes that follow.

Example C.7. A Sample Debugging Session

```
❶ $ CC/DEBUG/NOOPTIMIZE POWER
 $ LINK/DEBUG  POWER
 $ RUN  POWER
%DEBUG-I-NOGLOBALS, some or all global symbols not accessible

       OpenVMS I64 Debug64 Version E8.0

❷ %DEBUG-I-INITIAL, Language: C, Module: 'POWER'
%DEBUG-I-NOTATMAIN, Type GO to reach MAIN program

❸ DBG>  set break  %LINE 13
❹ DBG>  go
break at routine POWER\main
     12:     i = 2;
 DBG>  go
❺ break at POWER\main\%LINE 13
❻     13:    j = power(i);
❼ DBG>  step/into
❽ stepped to routine POWER\power
     16: int j;
 DBG>  step
 stepped to POWER\power\%LINE 18
     18:    return (j * j);
❾ DBG>  examine J
❿ %DEBUG-W-NOSYMBOL, symbol 'J' is not in the symbol table
 DBG>  examine j
⓫ POWER\power\j:  2
 DBG>  step
 stepped to POWER\main\%LINE 13+46
     13:    j = power(i);
 DBG>  step
 stepped to POWER\main\%LINE 14
     14: }
 DBG>  examine j
⓬ POWER\main\j:  4
 DBG>  go
```

```
❸ %DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful
                          completion'
❹ DBG>  exit
 $
```

Key to *Example C.7, "A Sample Debugging Session"*:

❶ To execute a program with the debugger, you must compile and link the program with the /DEBUG qualifier. The VSI C compiler compiles the source file with the /DEBUG=TRACEBACK qualifier by default. However, unless you compile your program with the /DEBUG qualifier, you cannot access all of the program's variables. Use the /NOOPTIMIZE qualifier to turn off compiler optimization that might interfere with debugging.

❷ The OpenVMS Image Activator passes control to the debugger on execution of the image. The debugger displays the current programming language and the name of the object module that contains the `main` function, or the first function to be executed. Remember that the linker converts the names of object modules to uppercase letters.

❸ You enter debugger commands at the following prompt:

```
DBG>
```

The SET BREAK command defines a point in the program where the debugger must suspend execution. In this example, SET BREAK tells the debugger to stop execution before execution of line number 13. After the debugger processes the SET BREAK command, it responds with the debugger prompt.

❹ The GO command begins execution of the image.

❺ The debugger indicates that execution is suspended at line 13 of the `main` function. The debugger specifies sections of the program by displaying the object module it is working in, delimited by a backslash character (\), followed by the name of the C function. The linker converted the name of the object module to uppercase letters but the debugger specifies the name of the function exactly as it is found in the source text.

❻ The debugger displays the line of source text where it suspended execution. Refer to the source code listing in *Example C.6, "Debugging Sample Program POWER.C"* to follow the debugger as it steps through the lines of the program in this interactive debugging example.

❼ The STEP/INTO command executes the first executable line in a function. The STEP command tells the debugger to execute the next line of code, but if the next line of code is a function call, the debugger will not step through the function code unless you use the /INTO qualifier. Use STEP/INTO to step through a user-defined or C RTL function.

❽ When stepping through a function, the debugger specifies line numbers by displaying the object module, the C function, and %LINE followed by the line number in the source text, each delimited by a backslash. The code at that line number is then displayed.

❾ The EXAMINE command displays the contents of a variable.

❿ The debugger does not recognize the variable `J` as existing in the scope of the current module.

⓫ Because the debugger supports the case sensitivity of C variables, variable `j` exists but variable `J` does not. Refer to *Example C.6, "Debugging Sample Program POWER.C"* to review the program variables.

In response to the EXAMINE command, the debugger displays the value of the variable `j` (2).

⓬ The value of variable `j` in function `main` is different from the local variable `j` in the `power` function. The `power` function executes properly, returning the value 2^2 (4).

⓭ When execution is completed, the debugger displays the execution status (successful, in this example).

⓮ The EXIT command terminates the debugging session and returns to the DCL prompt.

C.2. OpenVMS Text Processing Utility

The OpenVMS Text Processing Utility (TPU) (provided with the OpenVMS operating system) is a high-performance, programmable utility. TPU provides a number of special features, such as multiple buffers and windows, definable keys and key sequences, a procedural language, and a callable interface.

TPU serves as a base on which to layer other text processing applications, for example, text editors. The Extensible VAX Editor (EVE) is the editor provided with TPU. To invoke EVE, enter the following command at the DCL prompt:

```
$ EDIT/TPU USER.C
```

To exit from EVE, press the Do key to get the Command: prompt. If you want to save modifications to your file, enter the EXIT command. If you do not want to save the file or any modification to the file, enter the QUIT command.

C.3. Language-Sensitive Editor and the Source Code Analyzer

The Language-Sensitive Editor (LSE) and the Source Code Analyzer (SCA) must be purchased separately from the OpenVMS operating system. LSE is a text editor intended specifically for software development. SCA is an interactive tool for program analysis.

These products are closely integrated; generally, SCA is invoked through LSE. LSE provides additional editing features that make SCA program analysis more efficient. In addition, LSE and SCA, in conjunction with the VSI C compiler, provide a set of new enhancements supporting source code design and review.

In addition to text editing features, LSE provides the following software development features:

- Formatted language constructs, or templates, for most VSI programming languages, including VSI C. These templates include the keywords and punctuation used in source programs, and use placeholders to indicate locations in the source code where additional text is optional or required.
- Commands to compile, review, and correct compilation errors from within the editor.
- Integration with the Code Management System (CMS) for OpenVMS Systems. You can enter CMS commands from within the editor to make source file management more efficient.

SCA performs the following types of program analysis:

- Cross-referencing, which supplies information about program symbols and source files.

- Static analysis, which provides information on how subprograms, symbols, and files are related.

LSE and SCA together, in conjunction with VSI language compilers, provide the following software design features:

- Pseudocode support, which includes a new LSE placeholder for delimiting pseudocode. Pseudocode is text that describes algorithms or design decisions. This feature allows you to write source code in shorthand, returning later to fill in code details.
- Placeholder processing, in which language compilers accept LSE placeholders and pseudocode as valid program elements during compilation. This feature allows you to test the validity of algorithms while programs are still in shorthand form.
- Comment processing, which includes design comment information in the SCA library. SCA performs cross-referencing and static analysis on this information in response to user queries.
- View support, which provides a reverse-design facility. LSE commands compress program code into overview line summaries. If you choose to edit these overview lines, the modifications you make are reflected in the program code.
- A report tool, callable through LSE, that can print views, standard design reports, and customized reports.

C.3.1. Preparing an SCA Library

SCA stores data generated by the VSI C compiler in an SCA library. The data in an SCA library contains information about all symbols, modules, and files encountered during a specific compilation of the source. You must prepare this library before you enter LSE to invoke SCA. This preparation involves the following steps:

1. Create an OpenVMS directory for your SCA library. For example:

   ```
   $ CREATE/DIRECTORY PROJ:[USER.LIB1]
   ```

2. Initialize and set the library with the SCA CREATE LIBRARY command. For example:

   ```
   $ SCA CREATE LIBRARY [.LIB1]
   ```

   If you have an existing SCA library that has been initialized, you make its contents visible to SCA by setting it with the SCA SET LIBRARY command. For example:

   ```
   $ SCA SET LIBRARY [.EXISTING_SCA_LIBARAY]
   ```

   A message appears in the message buffer, at the bottom of your screen, indicating whether your SCA library selection succeeded.

3. Direct the VSI C compiler to generate data analysis files by appending the /ANALYSIS_DATA qualifier to the CC command. For example:

   ```
   $ CC/ANALYSIS_DATA PG1,PG2,PG3
   ```

   This command line compiles the input files PG1.C, PG2.C, and PG3.C and generates corresponding output files for each input file, with the file types OBJ and ANA. VSI C puts these files in your current default directory.

4. Load the information in the data analysis files into your SCA library with the SCA LOAD command. For example:

```
$ SCA LOAD PG1,PG2,PG3
```

This command loads your library with the modules contained in the data analysis files PG1.ANA, PG2.ANA, and PG3.ANA.

5. Once the SCA library has been prepared, enter LSE to begin an SCA session. Within this context, the integration of LSE and SCA provides commands that can be used only within LSE.

C.3.2. Starting and Terminating an LSE or an SCA Session

To invoke LSE, enter the following command at the DCL prompt:

```
$ LSEDIT USER.C
```

To end an LSE session, press Ctrl/Z to get the LSE> prompt. If you want to save modifications to your file, enter the EXIT command. If you do not want to save the file or any modification to the file, enter the QUIT command.

To invoke SCA from the LSE prompt, enter the SCA command that you want to execute using the following syntax:

```
LSE> command [parameter] [/qualifier...]
```

To invoke SCA from the DCL command line for the execution of a single command, use the following syntax:

```
$ SCA command [parameter] [/qualifier...]
```

If you have several SCA commands to invoke, you might want to first invoke the SCA subsystem and then enter SCA commands:

```
$ SCA
SCA> command [parameter] [/qualifier...]
```

Typing EXIT (or pressing Ctrl/Z) ends an SCA subsystem session and returns you to the DCL level.

C.3.3. Programming Language Placeholders and Tokens

The LSE language-sensitive features simplify the tasks of developing and maintaining software systems. These features include language-specific placeholders and tokens, aliases, comment and indentation control, and templates for subroutine libraries.

You can use LSE as a traditional text editor. In addition, you can use the power of LSE's tokens and placeholders to step through each program construct and supply text for those constructs that need it.

Placeholders are markers in the source code that indicate where you can provide program text. These placeholders help you to supply the appropriate syntax in a given context. You do not need to type placeholders; they are inserted for you by LSE. Placeholders are surrounded by brackets or braces and at (@) signs.

Placeholders are either optional or required. Required placeholders, indicated by braces ({}), represent places in the source code where you must provide program text. Optional placeholders, indicated by

brackets ([]), represent places in the source code where you can either provide additional constructs or erase the placeholder.

You can move forward or backward from placeholder to placeholder. In addition, you can delete or expand placeholders as needed.

Tokens typically represent keywords in VSI C. When expanded, tokens provide additional language constructs. You can type tokens directly into the buffer. You use tokens in situations, such as modifying an existing program, where you want to add additional language constructs and there are no placeholders. For example, typing IF and entering the EXPAND command causes a template for an IF construct to appear on your screen. You can also use tokens to bypass long menus in situations where expanding a placeholder, such as {@statement@}, will result in a lengthy menu.

You can use tokens to insert text when editing an existing file by typing the name for a function or keyword and entering the EXPAND command.

LSE provides commands for manipulating tokens and placeholders. *Table C.1, "Commands to Manipulate Tokens and Placeholders"* shows these commands and their default key bindings.

Table C.1. Commands to Manipulate Tokens and Placeholders

Command	Key Binding	Function
EXPAND	Ctrl/E	Expands a placeholder.
UNEXPAND	PF1-Ctrl/E	Reverses the effect of the most recent placeholder expansion.
GOTO PLACEHOLDER/FORWARD	Ctrl/N	Moves the cursor forward to the next placeholder.
GOTO PLACEHOLDER/REVERSE	Ctrl/P	Moves the cursor backward to the next placeholder.
ERASE PLACEHOLDER/FORWARD	Ctrl/K	Erases a placeholder.
UNERASE PLACEHOLDER	PF1-Ctrl/K	Restores the most recently erased placeholder.
[Enter] **[Return]**	{ENTER | RETURN}	Selects a menu option.

To display a list of all the defined tokens provided by VSI C, enter the following LSE command:

```
LSE> SHOW TOKEN
```

To display a list of all the defined placeholders provided by VSI C, enter the following LSE command:

```
LSE> SHOW PLACEHOLDER
```

To put either list into a separate file, first enter the appropriate SHOW command to put the list into the $SHOW buffer. Then enter the following commands:

```
LSE> GOTO BUFFER $SHOW
LSE> WRITE filename
```

To obtain a hard copy of the list, use the PRINT command at DCL level to print the file you created.

To obtain information about a particular token or placeholder, specify a token name or placeholder name after the SHOW TOKEN or SHOW PLACEHOLDER command.

C.3.4. Compiling Source Code

To compile your source code and to review compilation errors without leaving the editing session, use the LSE commands COMPILE and REVIEW. The COMPILE command issues a DCL command in a subprocess to invoke the VSI C compiler. The compiler then generates a file of compile-time diagnostic information that LSE uses to review compilation errors. The diagnostic information is generated with the /DIAGNOSTICS qualifier that LSE appends to the compilation command.

For example, if you enter the COMPILE command while in the buffer USER.C, the following DCL command is executed:

```
$ CC USER.C/DIAGNOSTICS=USER.DIA
```

LSE supports all the VSI C compiler's command qualifiers as well as user-supplied command procedures.

The REVIEW command displays any diagnostic messages that result from a compilation. LSE displays the compilation errors in one window and the corresponding source code in a second window. This multiwindow capability allows you to review your errors while examining the associated source code.

To compile a VSI C program that contains placeholders and design comments, include the following qualifiers to the COMPILE command:

```
LSE> COMPILE $/ANALYSIS_DATA/DESIGN
```

The /ANALYSIS_DATA qualifier generates a data analysis file containing source code analysis information. This information is provided to the SCA library.

The /DESIGN qualifier instructs the compiler to recognize placeholders and design comments as valid program elements. If the /ANALYSIS_DATA qualifier is also specified, the compiler includes information on placeholders and design comments in the data analysis file.

C.3.5. LSE Examples

The following examples show the expansions of VSI C tokens and placeholders. The intent is to show the formats and guidelines that LSE provides, not to fully expand all tokens and placeholders. An arrow (→) indicates where in the example an action occurred.

To invoke LSE and the VSI C language, use the following syntax:

```
LSEDIT [/qualifier…] filename.C
```

C.3.5.1. Compilation Unit

When you use the editor to create a new VSI C program, the initial string `{@compilation unit@}` appears at the top of the screen:

```
->      {@compilation unit@}
        [End of file]
```

Use Ctrl/E to expand this initial string. The following is displayed:

```
->      [@#module@]
        [@module level comments@]
```

```
        [@include files@]
        [@macro definitions@]

        [@preprocessor directive@]...

        [@data type or declaration@]...;

        [@function definition@]...;
```

C.3.5.2. Preprocessor Lines

Erase the `[@#module@]`, `[@module level comments@]`, `[@include files@]`, and `[@macro definitions@]`. The cursor is then positioned on `[@preprocessor directive@]`. Expand `[@preprocessor directive@]` to duplicate it and display a menu. Then select the `#include` option:

1. Use the up and down arrows on the keypad to position the displayed selection arrow next to `#include`.
2. Press Return.

The following display results:

```
->      #include
        [@preprocessor directive@]...

        [@data type or declaration@]...;

        [@function definition@]...;
```

After selecting the `#include` option, another menu appears that lists the types of `#include` statements. Select the option `#include {@module name@}`. Your display now looks like this:

```
->      #include {@module name@}
        [@preprocessor directive@]...

        [@data type or declaration@]...;

        [@function definition@]...;
```

Type the value `stdio` over the placeholder `{@module name@}`.

Experiment with the LSE editor to expand other placeholders, such as `[@data type or declaration@]`, `[@function definition@]`, and so on.

C.4. CDD/Repository

CDD/Repository is an optional OpenVMS software product available under a separate license. The CDD/Repository product allows you to maintain shareable data definitions (language-independent structure declarations) that are defined by a data or repository administrator.

Note

CDD/Repository supports both the Common Data Dictionary and CDD/Plus interfaces. Older dictionary versions need to be converted to repository (CDD/Repository) format using a supplied conversion utility. For detailed information about CDD/Repository, see the CDD/Repository documentation.

C.4.1. Using CDD/Repository

CDD/Repository data definitions are organized hierarchically in the same way files are organized in directories and subdirectories. For example, a repository for defining personnel data might have separate directories for each employee type.

Descriptions of data definitions are entered into the dictionary in a special-purpose language called CDO (Common Dictionary Operator, which replaces the older interface called CDDL, Common Data Dictionary Language).CDD/Repository converts the data descriptions to an internal form—making them independent of the language used to access them—and inserts them into the repository.

To extract data definitions from CDD/Repository, include the `#pragma dictionary` preprocessor directive in your VSI C source program. If the data attributes of the data definitions are consistent with VSI C requirements, the data definitions are included in the VSI C program during compilation. See *Section 5.4.3, "#pragma dictionary Directive"* for information about using `#pragma dictionary`.

CDD/Repository data definitions, in the form of VSI C source code, appear in source program listings if you specify the /SHOW=DICTIONARY qualifier on the CC command line.

The advantage in using CDD/Repository instead of VSI C source for structure declarations is that CDD/Repository record declarations are language-independent and can be used with several supported OpenVMS languages.

C.4.2. Accessing CDD/Repository from VSI C Programs

A repository or data administrator uses CDO to create repositories, define directory structures, and insert record and field definitions into the repository. Many repositories can be linked together to form one logical repository. If the paths are set up correctly, users can access definitions as if they were in a single repository regardless of physical location.

CDO also creates the record paths. Once established, records can be extracted from the repository by means of the `#pragma dictionary` preprocessor directive in VSI C programs. At compile time, the record definition and its attributes are extracted from the designated repository. Then the compiler converts the extracted record definition into a VSI C structure declaration and includes it in the object module.

The `#pragma dictionary` preprocessor directive incorporates CDD/Repository data definitions into the VSI C source file during compilation. The `#pragma dictionary` directive can be embedded in a VSI C structure declaration. See *Section 5.4.3, "#pragma dictionary Directive"* for sample usage of `#pragma dictionary`.

C.4.3. Support for CDD/Repository Data Types

CDD/Repository supports all OpenVMS data types. VSI C can translate all the OpenVMS data types when they are declared in CDD/Repository records. Data types that do not occur naturally in the VSI C language are handled in the following way:

- VSI C never attempts to approximate a data type that is not supported by the C language.
- Instead of approximating a data type, VSI C uses its own structure data type to represent all types (except for excessively long bit strings) not supported by the C language; specifically, VSI C creates structures of arrays of type `char` that are large enough to represent the data structure.

- Bit strings (aligned or unaligned) can be up to 32 bits long, as defined by the VSI C language. Bit strings longer than 32 bits are broken into increments of 32-bit strings or smaller so that the structure is correct with respect to size. However, the long bit string cannot be accessed as one unit.
- All row-major arrays are represented as zero-origin arrays of the appropriate size. An informational message is issued if the record description specifies nonzero-origin dimension bounds. The compiler adjusts the upper bound appropriately to maintain the correct number of elements relative to a lower bound of zero. Column-major arrays are converted to one-dimensional arrays containing the same total number of elements.

The compiler applies various consistency checks to the record attributes extracted from CDD/Repository, particularly the field data-type attributes. An error message is issued when a record description does not pass the consistency checks. An informational message is issued when VSI C is confronted with facility-independent attributes that are not supported. An error message is issued when an attribute that is required by VSI C is not present, even if the attribute is optional in CDD/Repository record protocol.

The compiler synthesizes names for unnamed and filler fields. If CDD/Repository does not specify a name and a name is required by the syntax of the VSI C language, the compiler synthesizes the name cc_cdd$_unnamed_nnnnn. When CDD/Repository specifies a filler or a name that VSI C does not support, the compiler synthesizes the name cc_cdd$_filler_#nnnnn, which includes the pound sign character (#). The string nnnnn represents a unique integer. The # is not a valid character in an identifier, so you cannot reference these fields.

Unsupported data types are mapped into VSI C as structures of character arrays of the appropriate size. The declaration of these data types uses the following format:

```
struct { char Cname [s]; } CDDname;
```

The CDDname is the name of the member in the CDD/Repository record. Cname is an identifier of the form cc_cdd_$_unsupported_#nnnnn, where nnnnn is a unique integer, and s is the size of the data item, in bytes.

VSI C generates `variant_struct` or `variant_union` declarations for unnamed CDD/Repository structures and unions, so you do not have to specify these references.

Table C.2, "Mapping Between CDD/Repository and VSI C Data Types" summarizes the mapping between CDD/Repository data types and VSI C data types.

Table C.2. Mapping Between CDD/Repository and VSI C Data Types

CDD/Repository Data Type	C Data Type
Unspecified	Unsupported
Unsigned byte	`unsigned char`
Unsigned word	`unsigned short`
Unsigned longword	`unsigned int`
Unsigned quadword	Unsupported
Unsigned octaword	Unsupported
Signed byte	`char`
Signed word	`short`

CDD/Repository Data Type	C Data Type
Signed longword	`int`
Signed quadword	Unsupported
Signed octaword	Unsupported
F_floating	`float`[1]
D_floating	`double`[1]
G_floating	`double`[1]
H_floating	Unsupported
F_floating complex	Unsupported
D_floating complex	Unsupported
G_floating complex	Unsupported
H_floating complex	Unsupported
Text	`char` [`n`]
Varying text[2]	Unsupported
Numeric string:	Unsupported
Unsigned	Unsupported
Left separate	Unsupported
Left overpunch	Unsupported
Right separate	Unsupported
Right overpunch	Unsupported
Zoned sign	
Packed decimal string	Unsupported
Bit	Bit field[3]
Bit unaligned	Bit field[3]
Date and time	Unsupported
Date	Unsupported
Virtual field	Ignored
Varying string[2]	Unsupported

[1]If the specification of the /FLOAT or /[NO]G_FLOAT qualifier conflicts with the data type of the CDD/Repository record member, an informational message is issued and the member is represented as `struct { char [8]}` instead of `double`.

This would occur if the data type of the CDD/Repository record member is D_floating, and G_floating format (the default) was specified on the CC command line; or if the data type of the record member is G_floating, and either D_floating or IEEE_floating (Alpha only) was specified on the command line; or if the data type of the record member is F_floating, and IEEE_floating (Alpha only) was specified on the command line.

[2]For these data types, the length of the structure is two bytes longer than the string to allow for the length field.
[3]A message is issued if the bit-string length is greater than 32.

Appendix D. VSI C Compiler Messages

This appendix lists the VSI C compiler diagnostic messages.

For each message, this appendix gives the mnemonic and the message text, an explanation of the message, and suggested actions to be taken to avoid the message. For more information about the format of compiler diagnostic messages, see *Section 1.3.5, "Compiler Diagnostic Messages"*.

To display a particular compiler message online, enter the following command:

```
$HELP CC MESSAGE mnemonic(Alpha, I64)
$ HELP CC/DECC MESSAGE mnemonic(VAX only)
```

To display a list of all compiler message mnemonics, enter the following command:

```
$ HELP CC MESSAGE(Alpha, I64)
$ HELP CC/DECC MESSAGE(VAX only)
```

Some messages substitute information from the program into the message text. In this appendix, the portion of the text to be substituted is shown in *italics*.

Often, the same message is issued in different contexts within a program. In this appendix, the message context is indicated by the italicized word *context* within the message. The actual message issued by the compiler will contain one of the following phrases substituted for *context*:

- In this declaration,
- In the initializer for,
- In the declaration of "*name*",
- In the definition of the function "*name*",
- In the declaration of an unnamed object,
- In this statement,

You can control the messages issued with the /[NO]WARNINGS command-line qualifier (*Section 1.3.4, "CC Command Qualifiers"*) and the `#pragma message` preprocessor directive (*Section 5.4.14, "#pragma message Directive"*).

DECDEC, In this declaration,

DECINITVR, In the initializer for,

DECNAMDEC, In the declaration of "*name*",

DECNAMFUNDEF, In the definition of the function "*name*",

DECUNDEC, In the declaration of an unnamed object,

PASSTA, In this statement,

ABSTRACTDCL, Invalid abstract declarator.

Description

An identifier was encountered in an abstract declarator. An abstract declarator is used to specify a type only and must not contain an identifier that specifies a declarator.

User Action

Correct the abstract declarator.

ADDRARRAY, *context*& before array "*expression*" is ignored.

Description

In certain modes, VSI C will ignore an address-of operator used on an entire array. This is for compatibility with other compilers that have this behavior.

User Action

Remove the address-of operator.

ADDRCONSTEXT, *context*"*name*" does not have a constant address, but occurs in a context that requires an address constant. This is an extension of the language.

Description

The C89 standard requires that an initializer for a pointer-type member of an automatic aggregate or union-type object have an initializer that is an address constant. Other C compilers might not successfully compile a program that uses this extension.

User Action

Be aware of this if you wish to port the program.

ADDRESSOFVOID, *context*taking the address of a void type is a language extension.

Description

The VSI C compiler will allow taking the address of a void type for compatibility with other compilers. This is an extension to the standard. Other compilers may reject this.

User Action

Be aware of this if you plan to port this source to another compiler.

ADDRSUBCONST, *context*accepting the expression "*expr*" as a constant is a language extension.

Description

In many cases VSI C accepts the subtraction of two addresses within the same array or struct/union as a constant. The C standard does not consider such an expression to be a constant. Therefore, this program does not conform to the standard and may be rejected by other compilers.

User Action

Change the expression to be a constant.

ALIGNCONFLICT, *context*the address "*expr*" has alignment of *align* which is less than the alignment requirements of the destination pointer. Dereferencing the destination pointer may cause an alignment fault.

Description

The compiler has detected a situation where a pointer to an aligned data type is being assigned an address that may not be properly aligned. A later dereference of this pointer could cause an alignment fault.

User Action

There are a number of possible actions. The best is to correct the condition that is causing the source to have the wrong alignment, as access to an unaligned data structure involves additional run-time overhead. Other options would be to modify the declaration of the destination pointer such that its referenced type has the __unaligned type qualifier, or use the compiler option that tells the compiler to assume all pointer references are unaligned. It is also possible to cast the source to the destination type to silence this message. However, that solution will not correct any unaligned access.

ALIGNCONFLICT1, *context*the address "*expr*" has alignment of *align* which is less than the alignment requirements of the pointer type it is cast to. Dereferencing the resulting pointer may cause an alignment fault.

Description

The compiler has detected a situation where an address is being cast to a pointer type with a greater alignment requirement than the type of the address expression implies. A later dereference of this pointer type value could cause an alignment fault.

User Action

There are a number of possible actions. The best is to correct the condition that is causing the source to have the wrong alignment, as access to an unaligned data structure involves additional run-time overhead. Other options would be to change the type of the pointer used in the cast such that its referenced type has the __unaligned type qualifier, or use the compiler option that tells the compiler to assume all pointer references are unaligned. It is also possible to cast the address expression to (void *) before casting it to the specified type to silence this message. However, that solution will not correct any unaligned access.

ALIGNCONST, Integer constant alignment *number* is not necessarily supported on all platforms.

Description

Although the specified alignment value is valid on this system, it might not be valid on other systems. For example, 16 is a valid alignment value on Alpha systems but would not be valid on VAX systems.

User Action

Be aware of this potential portability issue.

ALIGNEXT, _align is a language extension.

Description

The _align storage class modifier is a language extension of VSI C. Other C compilers might not successfully compile a program that uses the extension.

User Action

Be aware of this extension if you wish to port the code.

ALIGNPOP, This "restore" has underflowed the member alignment's stack. No corresponding "save" was found.

Description

The member_alignment stack, managed by the #pragma member_alignment and #pragma environment directives, contains more restores than saves. This could signify a coding or logic error in the program.

User Action

Make sure each restore has a corresponding save.

ALREADYTLS, The identifier "*name*" has already appeared in an omp threadprivate directive.

Description

The same identifier appears more than once in a single omp threadprivate directive, or appears in more than one omp threadprivate clause.

User Action

Remove the duplicate identifiers

ANSIALIASCAST, *context*a pointer to *type1* is being cast to a pointer to *type2*. Using ANSI aliasing rules, the compiler may subsequently assume that the two pointer types are pointing to different storage locations.

Description

The C standard allows a compiler to assume that these two pointer types will point to different storage locations. The compiler will make this assumption whenever ansi aliasing is enabled on the command line, either directly or via another switch. The cast in itself does not violate aliasing rules, e.g. you might cast the pointer value back to an allowed type before you use it to access memory. But the compiler cannot generally determine whether or not you do that. If your code accesses the memory designated by this pointer value using both of these pointer types, you may get unexpected results when ansi aliasing is enabled.

User Action

Casting through pointer to void will silence this message. But if the end result is that the same memory still gets accessed through different types that are not permitted under the aliasing rules, you may still get unexpected results. If compiling without ansi aliasing corrects the behavior of your program, your code almost certainly violates the aliasing rules in a way that the compiler cannot detect.

ARGADDR, *context*taking the address of the constant expression "*expression*" in an argument list is a language extension.

Description

The VSI C compiler will allow the address of a constant to be passed as an argument to a function call. This is an extension to standard C. Other C compilers might not successfully compile a program that uses this extension.

User Action

Assign the constant to a variable, and pass the address of the variable.

ARGLISGTR255, *context*the function call specifies an argument list whose length exceeds maximum specified by the calling standard. Any use of va_count by the called function will be wrong.

Description

The OpenVMS calling standard uses a byte-sized field to specify the size of the argument list. The argument list to this function call requires more storage than can be represented in this size. As a result, any use of va_count in the called function will return inaccurate information.

User Action

Either reduce the size of the argument list, or do not use va_count in the called function.

ARGSIZE, *context* the argument being passed to this function is too small.

Description

A function parameter of array type has been declared with the keyword "static" in its outermost bound to indicate that the function may generate code that assumes that when it is called the actual argument will have at least as many elements as specified in the parameter declaration. The argument provided in this call has fewer array elements than specified in the parameter declaration with static bound.

User Action

Check the size of the argument passed to the function and/or modify or remove the static bound on the function parameter.

ARRAYBRACE, *context* a required set of braces is missing.

Description

The initializer for this array was not enclosed in braces. While some compilers allow this, standard C requires braces around the initializer.

User Action

Enclose the initializer in braces.

ARRAYLIMITSUP, *context* VSI C provides only limited support for array types larger than *n* bytes.

Description

This array type is larger than can be represented by size_t. While VSI C will allow a type declared to be this size, uses of the type are not fully supported and may cause unpredictable behavior.

User Action

Reduce the size of the array type. It may be possible to use a pointer type instead of a large array. The storage can still be accessed using array syntax.

ARRAYOVERFLOW, Integer overflow occurred when computing the size of an array type.

Description

An array type is larger than allowed on this platform.

User Action

Reduce the size of the array type. It may be possible to use a pointer type instead of a large array. The storage can still be accessed using array syntax.

ARRNOTLVALUE, *context*accepting a non-lvalue array in a subscript operator is an extension to the C89 standard.

Description

The C89 standard states that one of the operands to the subscript operator must be a pointer. However, the array used in this operator could not be converted to a pointer because it is not an lvalue. Therefore this code does not conform to the C89 standard and may not be accepted by other compilers. Note that the C99 standard allows this because all arrays are converted to pointers, not just lvalue arrays.

User Action

Be aware of this difference if you plan to port this source to another compiler.

ASMCOMEXP, Comma expected while processing *text* instruction

Description

The asm directive parser was expecting a comma, but one was not found.

User Action

Correct the asm directive.

ASMENDEXP, Semicolon or asm end expected while processing *text* instruction

Description

The asm directive parser was expecting a semicolon to end an instruction, but one was not found.

User Action

Correct the asm directive.

ASMFIMMDOTS, Floating point load-immediate instructions require a .s file

Description

Using a floating point load immediate instruction in this asm directive will require the compiler to produce an .s file and invoke the assembler to process this source.

User Action

Do not use floating point load immediate instructions in asm directives.

ASMFREGEXP, Float register expected while processing *text* instruction

Description

The asm directive parser was expecting a valid floating register, but one was not found.

User Action

Correct the asm directive.

ASMHINTDOTS, Hint on *text* instruction requires a .s file

Description

Using a hint in a transfer instruction in this asm directive will require the compiler to produce an .s file and invoke the assembler to process this source.

User Action

Do not use hints in asm directives.

ASMICONEXP, Integer constant expected while processing *text* instruction

Description

The asm directive parser was expecting a valid integer constant, but one was not found.

User Action

Correct the asm directive.

ASMIDEXP, Identifier expected while processing *text* instruction

Description

The asm directive parser was expecting an identifier, but one was not found.

User Action

Correct the asm directive.

ASMINSTEXP, Instruction mnemonic expected (found *text*)

Description

The asm directive parser was expecting an instruction mnemonic, but one was not found.

User Action

Correct the asm directive.

ASMLABEXP, Label expected while processing *text* instruction

Description

The asm directive parser was expecting a label, but one was not found.

User Action

Correct the asm directive.

ASMLABMULDEF, Multiple definitions of label in asm (*text*)

Description

The asm directive parser has detected the same label defined more than once.

User Action

Change one of the label names.

ASMLABUNDEF, Reference to undefined label in asm (*text*)

Description

The asm directive parser has detected a reference to an undefined label.

User Action

Correct the asm directive.

ASMLDGPDOTS, Unusual ldgp requires a .s file

Description

This indicates that a ldgp pseudo-instruction was encountered in an unusual place or with unusual arguments. The assembler will be invoked on the .s file.

User Action

Correct the asm directive.

ASMLPAREXP, Left paren expected while processing *text* instruction

Description

The asm directive parser was expecting a left paren, but one was not found.

User Action

Correct the asm directive.

ASMNOTAVAIL, In-line assembly code directive *name* is not available on this platform.

Description

In-line assembly code is not available on the IA64 platform.

User Action

See documentation for alternatives.

ASMNOTINST, *text* instruction is not supported in asms on *text*

Description

The asm directive parser does not recognizes a pseudo-opcode on this platform.

User Action

Correct the asm directive.

ASMNOTREG, *text* is not a register name on *text*

Description

The asm directive parser has noticed that a special register used in the directive is not valid on this platform.

User Action

Correct the asm directive.

ASMNOTSUP, Support for *text* (*text*) in asms is not implemented on *text*

Description

The asm directive parser does not support the feature in question on this platform.

User Action

Rewrite the asm so that the feature is not used.

ASMPALTRUNC, PALcode function has been truncated to *number*

Description

The asm directive call_pal instruction is followed by an integer beyond the range of call_pal values expected by the compiler.

User Action

Use a valid call_pal argument.

ASMRAWREG, *text* uses *text* before it is defined

Description

The asm directive parser has noticed that an instruction uses a register as a source before it is given a value.

User Action

Correct the asm directive.

ASMREGEXP, Fixed register expected while processing *text* instruction

Description

The asm directive parser was expecting a valid integer register, but one was not found.

User Action

Correct the asm directive.

ASMREGOVRLAPSC, Destination register overlaps input for *text* (software completion) instruction

Description

An asm directive contains an instruction that may require a software completion routine in case of a runtime exception. Such an instruction requires that the result register be different than any input register.

User Action

Modify the asm so that the destination register is different than the sources.

ASMRPAREXP, Right paren expected while processing *text* instruction

Description

The asm directive parser was expecting a right paren, but one was not found.

User Action

Correct the asm directive.

ASMSYMDOTS, Use of symbolic addresses with *text* instruction requires a .s file

Description

Using a symbolic operand in this asm directive will require the compiler to produce an .s file and invoke the assembler to process this source.

User Action

Do not use symbolic operands in asm directives.

ASMUNKNOWNARCH, Unknown architecture (*text*) specified in *text* assembler directive

Description

The asm directive parser has detected an unexpected argument to a .tune or .arch directive.

User Action

Correct the asm directive.

ASMUNKSETOPT, Unsupported or illegal .set option (*text*)

Description

The asm directive parser has detected an unexpected argument to a .set directive.

User Action

Correct the asm directive.

ASSERTFAIL, The assertion "*assertion*" was not true, *reason*.

Description

The expression in a #pragma assert non_zero(expression) directive was found to be zero.

User Action

Correct the condition that caused the expression to be zero.

ASSERTION, *text*

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

ASSIGNEXT, *context*relaxed struct or union type compatibility is a language extension.

Description

In certain modes, the compiler will allow assignments or comparisons between structs or unions of different types if their sizes are the same. This is an extension to standard C. Other C compilers might not successfully compile a program that uses this extension.

User Action

Recode the operation to use one of the memxxx run-time library functions.

ASSUMEONEELEM, The type of the tentatively-defined array "*name*" is incomplete at the end of the compilation unit. The compiler will assume one array element.

Description

The C standard requires that the type of all tentative definitions must be completed before the end of the compilation unit. For compatibility with some other C compilers, VSI C will give the array one element.

User Action

Complete the type.

AUTOEXTERNAL, *context*a storage class of "auto" or "register" is illegal at file scope.

Description

The storage classes auto and register can only be used in a declaration that appears inside a function. They cannot be used in a declaration at file scope.

User Action

Remove the storage class specifier or move the declaration inside a function body.

BADALIAS, Reference through restricted pointer *text* uses a pointer value based on different restricted pointer, *text*

Description

The C language requires that restricted pointers always point to different storage. The compiler has detected a case where an access using a restricted pointer is referencing memory pointed to a different restricted pointer. This may cause unexpected behavior.

User Action

Make sure restricted pointers point at unique storage.

BADALIGN, Invalid alignment boundary.

Description

The _align storage class modifier was given an invalid value. See documentation for valid values on each platform.

User Action

Supply a correct value or remove the _align storage class modifier.

BADANSIALIAS, This statement accesses an object *frag1*. The statement at *loc* accesses the same storage location *frag2*.

Description

The standard allows a compiler to assume that since these two statements use different types, these two statements reference different storage locations. The VSI C compiler does so whenever ansi aliasing is enabled. Since your code relies on these two statements accessing the same storage location you should disable ansi aliasing. If you do not do so, optimization may cause your program to behave unexpectedly.

User Action

Specify noansi_alias on the command line.

BADBOUNDCHK, *context*pointer arithmetic was performed more than once in computing an array element. The bounds checking code output by the compiler will only verify the "*expr*" expression.

Description

When an array is accessed using pointer arithmetic and run-time array bounds checking is enabled, the VSI C compiler is only able to output the checking code for the first pointer arithmetic operation performed on the array. This can result in an incorrect check if the resulting pointer value is again operated on by pointer arithmetic. Consider the expression a = b + c - d; where a is a pointer, b an array, and c and d integers. When bounds checking is enabled the compiler will output a check to verify that c within the bounds of the array. This will lead to an incorrect runtime trap in cases where c is outside the bounds of the array and c - d is not.

User Action

Recode the pointer expression so that the integer part is in parenthesis. This way the expression will contain only one pointer arithmetic operation. In the earlier example the expression would be changed to a = b + (c - d);

BADBOUNDS, *context*the array bounds are incorrectly specified.

Description

A multi-dimensional array declaration contains a missing dimension specifier in a dimension other than the first.

User Action

Correct the declaration.

BADBREAK, This break statement is not within a for, while, do, or switch statement.

Description

A break statement can only appear inside a for, while, do, or switch statement.

User Action

Remove the break statement, or replace it with a goto statement.

BADC99PRAGOP, Invalid syntax for the C99 _Pragma operator, its operands cannot be recognized.

Description

After macro expansion and whitespace has been removed, the C99 _Pragma keyword must be followed by exactly three tokens: left-parenthesis, string-literal (or wide-string), right-parenthesis. Any other sequence cannot be processed, and will likely produce other spurious compile-time diagnostics.

User Action

Correct the syntax, or compile in a language mode that does not recognize the C99_Pragma operator (e.g. if your code has used this reserved identifier for some other purpose).

BADCHARSINHDR, Illegal characters after header name.

Description

While processing an #include directive whose argument did not start with either a '<' or '"' character, the compiler encountered a character it did not expect. This most often occurs when the directive argument is a macro and there is an error during the expansion of that macro.

User Action

Correct the argument to the #include directive.

BADCMMNTPSTNG, Token concatenation with comments might not be portable – use ## operator.

Description

A macro body contains a comment between two tokens with no white space either before or after the comment. Older C compilers allowed this as a form of token pasting. This type of token pasting might not give the desired results with newer compilers.

User Action

Use the standard C form of token pasting by replacing the comment with the ## token pasting operator.

BADCOMLITTYPE, *context*the type "*type*" cannot be used to specify the type of a compound literal.

Description

The type of a compound literal must be an object type or an array of unknown size.

User Action

Use a valid type.

BADCOMPLEXTYPE, *context*"*spelling*" is an invalid complex type specifier.

Description

The valid complex type specifiers are float _Complex, double _Complex, and long double _Complex.

User Action

Use on of the valid complex type specifiers.

BADCONDIT, *context*a common type could not be determined for the 2nd and 3rd operands ("*true expression*" and "*false expression*") of a conditional operator.

Description

The types of the second and third operands of the conditional operator must conform to a set of rules that define what the type of the result of the conditional operator itself will be. If the types of these operands do not conform to those rules, the compiler cannot determine the type of the result, which is an error. Refer to the language documentation for a complete list of valid combinations of types for the second and third operands of the conditional operator.

User Action

Modify the conditional expression so that the types of the second and third operands conform to the language rules.

BADCONSTEXPR, Syntax error in constant expression.

Description

A preprocessing constant expression contained a syntax error. The preprocessor was expecting to find a constant value or a left parenthesis. The preprocessor will assume a value of zero was encountered.

User Action

Correct the preprocessing constant expression.

BADCONTINUE, This continue statement is not within a for, while, or do statement.

Description

A continue statement can only appear inside a for, while, or do statement.

User Action

Remove the continue statement, or replace it with a goto statement.

BADCONVSPEC, *context*this argument to *function name* contains a bad conversion specification "*incorrect conversion*" that will cause unpredictable behavior.

Description

The compiler has detected an illformed conversion specification (flags, width, precision, length modifier) or an unknown conversion specifier (not diouxefgcspn...) that will cause unpredictable behavior. This might not have been what you intended.

User Action

Review the documentation for this function and modify the conversion specification as appropriate.

BADDCL, The name "*name*" cannot be undefined.

Description

The code has tried to #undef a macro that is predefined by the C standard. This is not allowed. The #undef will be ignored.

User Action

Remove the #undef directive.

BADDECLSPEC, Invalid argument to __declspec. Valid arguments are "thread" or "__thread".

Description

The only valid arguments to the __declspec storage class modifier are "thread" or "__thread".

User Action

Either use one of the valid arguments, or remove the storage class modifier.

BADDEFARG, Bad argument for "defined" operator.

Description

The defined preprocessing operator was given an invalid argument. The operator expects an identifier optionally enclosed in parenthesis. The value of the operator is undefined.

User Action

Supply a valid argument to the preprocessing operator.

BADENUM, Invalid enumerator.

Description

While processing an enumerator list, the compiler was expecting to encounter an identifier, but it found something else instead.

User Action

Correct the program syntax.

BADENUMREDECL, *context*the enum "*tag*" cannot be given a type other than signed int because the tag was declared earlier at *where*.

Description

This enum tag would normally be given a type other than signed int because the enumeration constants used in the declaration exceed the range of signed int. The compiler cannot use the extended type because the enum tag was declared earlier, and given signed int type at that point.

User Action

Remove the earlier tag declaration.

BADEXPR, Invalid expression.

Description

An invalid expression was encountered.

User Action

Correct the program syntax.

BADFATCOMMENT, The compiler cannot recover.

Description

In certain cases, the compiler cannot proceed after an unterminated comment. In these cases this message will be issued. Note that this message is always output after the opencomment error has been output.

User Action

Terminate the comment before the end-of-file.

BADFBDAT, *text* contains invalid feedback data

Description

A feedback file contains data, but it was corrupt and could not be used.

User Action

Create a new feedback file.

BADFBFILE, Invalid feedback file: *text*

Description

The compiler was unable to read information from the specified feedback file.

User Action

Make sure the feedback file contains valid feedback information.

BADFBTYP, Unexpected file type for feedback file *text*

Description

The file specified in the -feedback option does not have the file type expected by the compiler.

User Action

Use a valid feedback file.

BADFLOATTYPE, *context*this floating point type "*type*" is not supported on this platform.

Description

The IEEE floating types __s_float and __t_float are not supported on the VAX platform.

User Action

Change the type to a floating type that is supported on VAX, or compile the application on a platform that does support IEEE floating.

BADFORMALPARM, This token may not appear in a formal parameter list.

Description

While processing the formal parameter list of a macro definition, the compiler encountered an invalid formal parameter specifier. The macro will be be defined and this token will ignored, but that may not have been what you intended.

User Action

Correct the formal parameter list so that it consists of a comma separated list of identifiers.

BADFORSTOCLS, The declaration in a for loop can only have storage class auto or register.

Description

The declaration in a for loop contains a storage class specifier other than auto or register. This is not allowed.

User Action

Correct the storage class.

BADFUNCSTOCLS, The storage class of function *name* cannot be *storage_class*. This storage class has been changed to 'extern'.

Description

The globalref storage class cannot be used with a function declaration. The compiler will use the storage class extern.

User Action

Remove the globalref storage class from the function declaration.

BADGLOBALTYPE, This declaration has type "*type*", which is invalid for a globalvalue. The extern_model strict_refdef will be used instead.

Description

An object with globalvalue storage class can only have a type of integer, enum, or pointer type. In other cases, the compiler will change the storage class from globalvalue to strict_refdef.

User Action

Change the data type to be one that is valid for a globalvalue.

BADHEADERNM, Invalid include file or header name specification.

Description

An #include directive was not followed by a valid argument. The directive will be ignored. The #include directive should be followed by either a file specification enclosed in angle brackets, a file specification enclosed in quotes, or an identifier that specifies a text module (OpenVMS only), or a macro to be expanded.

User Action

Supply a valid argument to the #include directive.

BADHEXCONST, Hex constant value too large.

Description

A hex constant used in a preprocessor directive is too large. The value of the constant will be undefined.

User Action

Decrease the value of the constant.

BADIDENTUCN, Invalid UCN encountered in an identifier.

Description

An identifier contained a Universal Character Name (UCN) that did not conform to the requirements of C99 Annex D for use of UCNs in identifiers.

User Action

Specify a valid UCN sequence.

BADIFDEF, An #ifdef or #ifndef is not followed by an identifier.

Description

An #ifdef or #ifndef preprocessing directive was not followed by an identifier. The compiler will consider the preprocessor argument to be an identifier that is not defined. Therefore, in these cases an #ifdef will always be FALSE, and an #ifndef will always be TRUE.

User Action

Supply a valid identifier to the directive.

BADIFNDEFARG, #ifndef argument is not an identifier.

Description

An #ifndef preprocessing directive was not followed by an identifier. The compiler will consider this to be a TRUE condition.

User Action

Supply a valid identifier to the directive.

BADINCLDIR, The #pragma include_directory must not appear after an #include directive or in a /FIRST_INCLUDE file after the first /FIRST_INCLUDE file has been processed. The directive will be ignored.

Description

There are several restrictions on the placement of the #pragma include_directory directive. It must not appear after any #include directive has been encountered. Also, if /FIRST_INCLUDE is specified on the command line, all #pragma include_directory directives must be placed in the first file in the /FIRST_INCLUDE list (if there is more than one in the list) or in the the main source before any #include directives (if there is only one file in the /FIRST_INCLUDE list).

User Action

Place the directive in a valid location.

BADINCLDIRSIZE, The include_directory string length must be at least one and must be less than *max*. The directive will be ignored.

Description

The #pragma include_directory directive does not support an empty string argument. Also the directory must not exceed the longest directory specification supported on this platform.

User Action

Specify a valid length string.

BADINCLUDE, An #include directive has illegal syntax.

Description

An #include directive was not followed by a valid argument. This message occurs when the argument starts with a '<' or '"' character, but does not end with a matching delimiter. In this case the compiler will add the matching delimiter to the end of the argument and process the directive normally.

User Action

Correct the argument to the #include directive.

BADLINEDIR, Missing argument for #line directive.

Description

An argument was not supplied to a #line preprocessing directive. This directive must be followed by a digit sequence that specifies the line number or a macro that expands to a digit sequence. The directive will be ignored.

User Action

Supply a valid argument to the directive.

BADLINEDIRTV, Illegal token in #line directive.

Description

A #line directive was followed by an invalid argument. The #line directive should be followed by either a digit sequence or a digit sequence followed by a string literal. The #line directive will be ignored.

User Action

Supply a valid argument to the #line directive.

BADLINKREG, Invalid register "*register*" for linkage pragma. Pragma is ignored.

Description

The compiler encountered bad register specifier in a #pragma linkage directive. The message should point at the offending specifier. The compiler will ignore the entire pragma.

User Action

Correct the directive.

BADLINNUM, Ignoring the line number for the #line directive – too small.

Description

A #line preprocessing directive specified a line value that is either zero or less than zero. This is not valid. The directive will be ignored.

User Action

Either remove the directive or supply a positive value to the line specifier.

BADLOCALE, The compiler could not set its locale to either the locale-specific native environment or the "C" locale.

Description

During start-up, the compiler was unable to set its locale. As part of its initialization, the compiler will issue the call setlocale(LC_ALL, ""). If this call fails, the compiler will try to issue the call setlocale(LC_ALL, "C"). If this call also fails, the compiler will issue this message and abort.

User Action

The best way to determine why the compiler is failing is to write a small program that contains the same library calls the compiler is making and then examine the return values.

BADMACROINLN, Illegal token from macro call in #line directive.

Description

A #line directive was followed by a macro whose expansion did not form a valid argument to the directive. The #line directive should be followed by either a digit sequence or a digit sequence followed by a string literal. The #line directive will be ignored.

User Action

Supply a valid argument to the #line directive.

BADMACRONAME, "*directive*" directive is not followed by an identifier and is being ignored.

Description

A #define or #undef preprocessing directive was not followed by an identifier. The first argument to these directives must be an identifier that specifies the macro to define or undefine. The compiler will ignore the directive.

User Action

Correct the argument to the preprocessing directive.

BADMBCOMMENT, An invalid multibyte character was encountered in a comment.

Description

An invalid multibyte character was found in a comment. While this will not affect the program execution, it might not have been what you intended.

User Action

Correct the multibyte character.

BADMCRORECURS, Recursive expansion of macro "*name*" exceeded *num* levels and was terminated.

Description

In certain cases, the compiler will allow a macro to be recursively expanded. In these cases, the compiler limits the level of the recursion to prevent the compiler from looping to the point where it consumes all available memory. When this level has been reached, this message is output.

User Action

Rewrite either the macro definition or the macro invocation so that the recursion ends before the compiler limit is reached. Note that the use of recursive macros is not a feature of the C standard, and most other C compilers will not support this.

BADMEMBER, Invalid member declaration.

Description

A struct or union contains an invalid member declaration. In most cases this error occurs when a semi-colon was omitted from the previous member declaration.

User Action

Correct the declaration.

BADMEMOFF, *context*multiple definitions of member "*name*" found with different offsets.

Description

In certain modes, the compiler will allow a struct or union reference whose right operand is not a member of the struct or union type of the left operand. This is allowed for compatibility with other compilers. However, in these cases the right operand must specify a member name that is declared with the same type and at the same offset in every struct or union type that declares it. This message is issued when the compiler finds member name it is looking for declared with a different offset in more than one struct or union type.

User Action

VSI recommends that the left operand or a struct or union reference specify a member that is a member of the type of the struct or union specified by the right operand. If this modification cannot be made then the member specified by the left operand must be declared at the same offset and with the same data type in all struct or union declarations that declare that member.

BADMEMTYP, *context*multiple definitions of member "*name*" found with different types.

Description

In certain modes, the compiler will allow a struct or union reference whose right operand is not a member of the struct or union type of the left operand. This is allowed for compatibility with other compilers. However, in these cases the right operand must specify a member name that is declared with the same type and at the same offset in every struct or union type that declares it. This message is issued when the compiler finds a member name it is looking for declared at the same offset but with different types in more than one struct or union type.

User Action

VSI recommends that the left operand or a struct or union reference specify a member that is a member of the type of the struct or union specified by the right operand. If this modification cannot be made then the member specified by the left operand must be declared at the same offset and with the same data type in all struct or union declarations that declare that member.

BADMODULEID, Invalid identifier found immediately following "#pragma module" or "#module" directive.

Description

The #pragma module or #module directive must be followed by an identifier that specifies the module name used by the linker.

User Action

Correct the directive.

BADMULTIBYTE, An invalid multibyte character was encountered *in type of construction*.

Description

An invalid multibyte character was encountered. The message will provide additional information about the location and attempted use of the character.

User Action

Correct the multibyte character.

BADNUM, *text* Qualifier value '*text*' is not an integer

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

BADOCTCONST, Octal constant value too large.

Description

An octal constant used in a preprocessor directive is too large. The value of the constant will be undefined.

User Action

Decrease the value of the constant.

BADOPCCAP, *text* instruction used is not in the selected instruction set

Description

The compiler has output an instruction that is not in the instruction set selected on the command line. One way this can happen is to compile a program which contains a floating point operation and specifying that no floating point instructions should be generated.

User Action

Either modify the source so the instruction will not be necessary, or use a different instruction set.

BADOPENBRACE, This open brace may be missing a close brace and causing the syntax error at *location*.

Description

This message is always output to the terminal after another syntax error. It is intended to provide the programmer with additional information that may identify the cause of the syntax error. This message may, or may not, provide useful information. In general, the more consistent the coding style in the source function, the more likely this message will be accurate.

User Action

Correct the program syntax.

BADPARSEDECL, In this declaration, "*id*" must specify a type.

Description

In processing a declaration, the type of the declarator has not been declared as a typedef.

User Action

Either declare the type as a typedef, or correct the spelling of the type specifier in this declaration.

BADPARSEPARAM, In this parameter list, "*param*" must either be a type or must be followed by a ",".

Description

In processing a function declaration, the compiler has found a case where the parameter list begins with two identifiers not separated by a comma and where the first identifier is not a type specifier. If this is an old-style declaration the two identifiers must be separated by a comma. If this is a prototype declaration, the first identifier must specify a type.

User Action

Correct the function parameter specifiers.

BADPPDIR, File ends in an unfinished pp directive.

Description

An unexpected end-of-file was encountered during a preprocessing directive.

User Action

Correct the directive.

BADPRAGMAARG, Unexpected or missing argument to #pragma *pragma name*. Pragma is ignored.

Description

An argument to a #pragma preprocessing directive is either missing or is not correct. The compiler will ignore the directive.

User Action

Correct the directive.

BADPRAGMAARG1, Unexpected token encountered in pragma. Found "*found*" when expecting *expecting*. The pragma will be ignored.

Description

While parsing a #pragma directive, the compiler has encountered something unexpected. The message will contain information about what the compiler was expecting as well as what it found.

User Action

Correct the offending directive.

BADPRAGMALINK, A bad linkage pragma was specified. Pragma is ignored.

Description

The compiler encountered a bad #pragma linkage directive. The error message should point to the place in the pragma that the compiler considers bad. The compiler will ignore the entire pragma.

User Action

Correct the directive.

BADPRAGNAMES, Invalid argument to the pragma names directive. Pragma is ignored.

Description

An invalid argument has been specified for the #pragma names preprocessing directive.

User Action

Correct the argument to the pragma.

BADPREFIX, Argument to extern_prefix is not a recognized keyword or a quoted string. Pragma is ignored.

Description

An invalid argument has been specified for the #pragma extern_prefix preprocessing directive. The directive expects either the identifiers "save", "__save", "restore", "__restore", or a string constant that specifies the external prefix to use. The compiler will ignore the pragma.

User Action

Correct the argument to the pragma.

BADPROTYP, Unexpected file type for profile file *text*

Description

The file specified in the -feedback option does not have the file type expected by the compiler.

User Action

Use a valid feedback file.

BADPTRARITH, *context*performing pointer arithmetic on a pointer to void or a pointer to function is not allowed. The compiler will treat the type as if it were pointer to char.

Description

Pointer arithmetic is not allowed on pointers to function or void types For compatibility with some other compilers, an output file is still created. The result produced will be the same as if the pointer were a pointer to char. This may or may not be compatible with other compilers that accept this syntax.

User Action

Cast the pointer type to a pointer to object type before performing the arithmetic.

BADREGISTER, *context*"*name*" has register storage class, but occurs in a context that precludes register storage. The storage class has been changed to auto.

Description

An object that was declared with register storage class has been referenced in a way that is not valid for a register. The most common example is taking the address of an object declared with register storage class. As certain array accesses also require taking the address of an array, this message can also be output for accessing the element of an array declared with register storage class. The compiler will change the storage class from register to auto.

User Action

Either remove the register storage class from the declaration, or change the reference to be one that is valid for objects with register storage class.

BADRETURNTYPE, *context*a function cannot return *type* type.

Description

A function return type cannot be an array or function type.

User Action

Correct the function declaration so that the return type is valid.

BADSEVERITY, The severity of message id *name* cannot be made less severe. The severity for this message was not changed.

Description

The severities of the compiler's error and fatal messages cannot be changed to a severity that is less severe. The compiler's fatal messages cannot be changed to any other severity. The compiler's error messages can only be changed to fatals.

User Action

Remove the pragma or compiler option that tried to change the severity.

BADSTATICCVT, *context*the address cannot be converted to the destination type.

Description

A static initialization tried to convert a link-time address to another type. However, the linker on this platform will not support such a conversion.

User Action

Rewrite the static initialization, or perform the initialization using runtime code.

BADSTDLINKAGE, If standard_linkage is used, it must be the only characteristic specified.

Description

The standard_linkage characteristic cannot be used with any other linkage characteristic.

User Action

Correct the pragma.

BADSTMT, Invalid statement.

Description

An invalid statement was encountered. The most common cause of this error is when a declaration appears after the first statement in a compound statement.

User Action

Correct the program syntax.

BADSTMT1, Invalid statement. This condition may have been caused by an open brace without a matching close brace. The compiler will attempt to identify open braces that might be missing a close brace.

Description

An invalid statement was encountered. This condition may have been caused missing close brace. This message is followed by some number of additional messages that attempt to identify

User Action

Correct the program syntax.

BADSUBSCRIPT, *context*an array subscript expression is either less than zero or greater than the largest value that can be represented by the size_t type.

Description

The compiler has detected an array subscript expression that is outside the bounds of any valid array. The array access might cause unpredictable behavior.

User Action

Specify a valid array subscript.

BADTARGMACRO, The target macro "*name*" does not match the compiler's target. This will likely cause incorrect code paths to be taken.

Description

On OpenVMS IA-64, some users have tried defining the macro __ALPHA explicitly using /DEFINE or a #define in a /FIRST_INCLUDE file as a quick way to deal with source code conditionals that assume that if __ALPHA is not defined then the target must be a VAX. Defining __ALPHA will cause many of the CRTL and other OpenVMS headers to take the wrong path for I64.

User Action

Remove any definitions of Alpha target macros, and if necessary correct the preprocessor conditionals that seemed to require an Alpha target macro to get the desired effect. E.g. change "#ifdef __ALPHA" to "#ifndef __VAX" or "#if defined(__ALPHA) || defined(__ia64)".

BADTKEN, Lexically invalid token.

Description

An invalid token was encountered in a preprocessing directive.

User Action

Correct the preprocessing directive.

BADUNKNOWNVLA, *context*a "*" bounds specifier is invalid. Using a "*" to specify a variable-length array of unknown size is only valid in declarations with function prototype scope.

Description

Using a "*" as a bounds specifier to designate a variable-length array with unknown size is only valid in declarations with function prototype scope.

User Action

Supply a valid bound specifier.

BADUNROLLVAL, The #pragma unroll directive takes a value from zero to 255. The value "*val*" is outside that range. The directive will be ignored.

Description

The value supplied to a #pragma unroll is outside the range allowed for the directive. The #pragma directive will be ignored.

User Action

Use a valid value for the unroll count.

BADUSELINK, A bad use_linkage pragma was specified. Pragma is ignored.

Description

The compiler encountered a bad #pragma use_linkage directive. The error message should point to the place in the pragma that the compiler considers bad. The compiler will ignore the entire pragma.

User Action

Correct the directive.

BADUSERMACRO, The name "*name*" cannot be a user-defined macro.

Description

The code has tried to #define either a macro that is predefined by the C standard or the DEFINED preprocessing keyword. This is not allowed. The #define will be ignored.

User Action

Remove the #define directive.

BADVASTART, *context*old-style parameter "*name*", with type that requires default argument promotion, cannot be used with va_start.

Description

It is invalid for the parameter specified in va_start to be one that requires default argument promotion.

User Action

The recommended fix is to recode the function definition to use a prototype-format definition. It is also possible to change the parameter declaration to use one of the default types, for example double.

BIFENABLED, The function "*routine name*" is a builtin function reserved to the compiler, and does not require a #pragma intrinsic. The function will continue to be treated as a builtin.

Description

A function identifier specified in a #pragma function intrinsic is the name of a builtin function. These functions cannot be explicitly enabled, they are always handled as builtin functions.

User Action

Remove the inappropriate use of the pragma.

BIFNEEDSSTD, *context*use of "*function*" is not allowed in a function with a non-standard linkage. This function was given the linkage "*name*" by a #pragma use_linkage directive.

Description

Certain built-ins that return information about a function call require that the function be called with standard linkage. Because this function appears in a #pragma use_linkage directive naming a linkage that specifies attributes other than standard_linkage, these builtins cannot be called from this function.

User Action

Use a standard linkage on this function, remove the calls to the builtins, or move them to a different function that is called with standard linkage.

BIFNOTAVAIL, Built-in function *name* is not available on this platform.

Description

This Alpha built-in function is not available on the IA64 platform.

User Action

See documentation for alternatives.

BIFPROTO, *context*the built-in function, "*name*", requires a prototype declaration from *filename*.

Description

Invoking a built-in function requires that the function be declared before it is invoked. This should be done by including the header file noted in the message.

User Action

Include the header file before the function is invoked.

BITARRAY, The CDD description for *name* specifies that it is an array of bitfields; It has been converted to a scalar bitfield.

Description

VSI C does not allow arrays of bitfields. The resulting C declaration will be a bitfield of the same total size as that specified in the CDD description.

User Action

If a bitfield type is acceptable, then no user action is necessary. If, however, the bitfield type is not acceptable, then the CDD description should be altered.

BITBADREP, *context*the bitfield type is not an integral type.

Description

A bitfield has been declared with a non-integral type. Standard C requires that all bitfields be declared with either int, unsigned int, or signed int type.

User Action

Change the type of the bitfield.

BITCONSTSIGN, *context*the integer constant "*constant*" does not have the same sign as the 1-bit bitfield it is being converted to.

Description

Either an unsigned 1-bit bitfield was assigned -1, or a signed 1-bit bitfield was assigned 1. This may not be what you intended.

User Action

Change the constant to be the appropriate sign.

BITFIELDSIZE, The CDD description for bitfield *name* specifies a size greater than 32; The excess is declared separately.

Description

VSI C does not allow individual bitfields larger than 32. As a result, a series of bitfields have been declared whose total size matches that of the CDD definition.

User Action

If the generated definitions are acceptable, then no user action is necessary. If, however, the generated definitions are not acceptable, then the CDD description should be altered.

BITNOTINT, *context*the bitfield type is not an int, signed int, unsigned int or _Bool.

Description

A bitfield has been declared with a type other than int, signed int, unsigned int or _Bool. This is not allowed by the C standard.

User Action

Change the declaration to use one of the allowed types or compile with a standard mode that allows this behavior.

BITWIDTH, *context*the bitfield width expression "*expression*" is outside the range *lower* to *upper*.

Description

A bitfield width specifier was either less than zero, or is greater than the number of bits in an int. In some modes, the compiler will assume a width specifier equal to the number of bits in an int.

User Action

Use a valid bitfield width specifier.

BITWIDTHTYP, *context*the bitfield width expression "*expression*" does not have an integral type.

Description

A bitfield width specifier does not have an integral type. A bitfield width specifier must be an integral constant expression.

User Action

Correct the width specifier.

BLOCKEXTVLA, *context*the block scope identifier "*name*" cannot be declared with a variably modified type because it has extern storage class.

Description

Only ordinary identifiers with block scope and without storage class extern, or ordinary identifiers with function prototype scope can be declared with a variably modified type.

User Action

Correct the declaration.

BLOCKINL, Block level declarations of inline functions are not allowed.

Description

In C99 standard, block level declaration of inline functions are prohibited.

User Action

Move the inline function declaration to file scope.

BLTINARGCNT, *context*an incorrect number of arguments were passed to the builtin function, "*function expression*".

Description

This message is output on OpenVMS systems when the number of arguments passed to the builtin function is not one.

User Action

Correct the call to the builtin function.

BLTINIMPLRET, *context*for the function "*name*", the implicit return type of "*type*" is not consistent with the expected type of "*type*". It will be treated as an ordinary implicitly defined external function.

Description

A function that could be handled internally by the compiler has not been declared, so an implicit declaration has been created for the function. The return value for the function is being used, and the implicit return type does not agree with what the compiler expected to see. In such cases, the function will not be handled internally, but will instead be called at run time in the usual manner. This could result in a performance loss, or possibly incorrect results if the implicit return type is incorrect.

User Action

If the function is intended to refer to the runtime library routine, the appropriate header file should be included in the source. Alternatively, a correct prototype could be provided privately in the source file. If the function is intended to be a replacement for the runtime library routine, disable the intrinsic version by specifying "#pragma function(function_name)" in the source file.

BOOLEXT, The _Bool data type is a new feature in the C99 standard. Other C compilers may not support this feature.

Description

This is a new language feature in C99. While having a standard specification for portability, the feature may not yet be available in all of the compilers you use.

User Action

Determine whether or not the use of this feature will cause portability problems for this code.

BOOLNA, The _Bool keyword is not supported in this language mode. It will be treated as an identifier in this compilation.

Description

Support for the _Bool keyword is only available in certain language modes. Support is not present when the compiler is in VAX C, K & R (common), or strict ANSI89 standard modes. In these language modes _Bool will be treated as an identifier.

User Action

Compile using one of the other compilation modes.

BOUNDADJ, The CDD description for *name* specifies non-zero-origin dimension bound(s); The bound(s) are adjusted to zero-origin.

Description

The CDD description specifies lower bounds(s) for an array that is non-zero. The resulting C definition will have the upper bound(s) adjusted for lower bound(s) of zero.

User Action

Verify that all subscript expressions are referencing the correct array element(s).

BOUNDNOTINT, *context*the array bound "*expression*" does not have an integral type.

Description

The compiler has encountered an array-bounds specifier that is not an integral type. Array-bounds specifiers must be positive integer constants.

User Action

Correct the array-bounds specifier

BUGCHECK, Compiler bugcheck. Submit a problem report with a problem description.

Description

An unexpected condition occurred in the compiler. This is most likely caused by a compiler bug.

User Action

Reduce the program that is causing the failure as much as possible. This often leads to a small test case. Please submit a problem report containing enough information for Engineering to reproduce the problem. The problem report should include the small test case.

CALLNEEDSFUNC, *context*"*expression*" is not a function.

Description

In what appears to be a function call, the expression denoting the the function to call is neither the identifier for a function nor an expression of type pointer to function.

User Action

Correct the expression denoting the function. If the expression is a simple identifier, perhaps a function-like macro definition is missing.

CANNOTREDEF, Cannot #define a macro that is currently expanding.

Description

The program is trying to #define the same macro it is currently expanding. The #define will be ignored.

User Action

Remove the #define, or move it after the expansion of the macro.

CANNOTUNDEF, Cannot #undef a macro that is currently expanding.

Description

The program is trying to #undef the same macro it is currently expanding. The #undef will be ignored.

User Action

Remove the #undef, or move it after the expansion of the macro.

CANTDISABLE, The message id *name* cannot be disabled.

Description

The compiler's error and fatal messages cannot be disabled.

User Action

Remove this message id from the list of messages being disabled on the command line or in the #pragma message line.

CANTMKRPSTORY, Attempt to create repository "*string*" for shortend names failed; OpenVMS status: *reason*.

Description

A compilation that used the /NAMES=SHORTENED qualifier could not open the repository used to store the shortened names. This could be because an invalid name was specified in the /REPOSITORY qualifier. The message will give additional information about the failure.

User Action

Correct whatever caused the failure.

CDDATTR, One or more field descriptions in this CDD record specify an attribute that is being ignored.

Description

The CDD description specifies an attribute that is not supported in VSI C. The attribute is ignored.

User Action

No action is required.

CDDBADID, An invalid identifier, *name*, is being ignored in the dictionary directive.

Description

An unexpected identifier follows the dictionary pathname in a dictionary preprocessing directive. The identifier is ignored.

User Action

Remove the invalid identifier(s) in the dictionary directive.

CDDEXT, #dictionary is a language extension.

Description

The #dictionary directive is an extension of VSI C on OpenVMS. The program might not compile with other compilers or on other platforms.

User Action

Be aware of this if you wish to port the program.

CDDPATH, A valid CDD pathname was not found. The CDD directive has been ignored.

Description

The #dictionary preprocessing directive was not followed by an argument. The directive must be followed by a character string that gives the path name of a CDD record, or a macro that expands to the path name of the record.

User Action

Supply a valid argument to #dictionary. VSI also recommends that the #dictionary preprocessing directive be replaced by the #pragma dictionary operator.

CDDTOODEEP, The attributes for the Common Data Dictionary record description *name* exceed the implementation's limit for record complexity.

Description

The CDD description specifies more attributes than the interface between the CDD and the compiler can handle.

User Action

Simplify the record description.

CHARCONST, Ill-formed character constant.

Description

An invalid character constant was encountered.

User Action

Correct the character constant.

CHAROVERFL, A character constant value requires more than sizeof(int) bytes of storage.

Description

A character constant is too long to fit in an int. The compiler will ignore the extra characters.

User Action

Remove the extra characters from the character constant.

CHKEXPAND, *number* integrity check error(s) after IL expansion of routine *text*

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

CHKINIT, *number* integrity check error(s) in initial IL & ST for module *text*

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

CHKOPT, *number* integrity check error(s) after *text* optimization phase for routine *text*

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

CLASSNOINIT, *context*the struct or union object "*name*" is uninitialized and has a const member.

Description

An object of struct or union type has a const member and has not been initialized. This might not have been what you intended. VSI recommends that you initialize all objects with the const attribute. The missing initializer will make this an invalid declaration in C++.

User Action

Initialize the struct or union object.

CLOSBRACKET, Missing "]".

Description

The compiler was expecting a closing bracket, but one was not found.

User Action

Correct the program syntax.

CLOSEBRACE, Missing "}".

Description

The compiler was expecting a closing brace, but one was not found.

User Action

Correct the program syntax.

CLOSECOMMENT, This unmatched comment delimiter is ignored.

Description

An unmatched comment delimiter (*/) is an illegal combination of unary indirection and binary division operators that would have caused your compilation to fail.

User Action

Remove the comment delimiter.

CLOSEPAREN, Missing ")".

Description

The compiler was expecting a closing parenthesis, but one was not found.

User Action

Correct the program syntax.

CMPPTRFUNVOID, *context*accepting the [in]equality comparison of a pointer to void and a pointer to function type is a language extension.

Description

Under the C standard, it is a constraint violation to perform an [in]equality comparison between a pointer to void and a pointer to function type. Therefore this code may not be accepted by other compilers.

User Action

Cast one of the pointers to the type of the other.

COLMAJOR, The CDD description for *name* specifies that it is a column-major array; It has been converted to a one-dimensional array.

Description

The VSI C compiler supports only row-major arrays. Therefore the column-major array description in the CDD has been converted to a one-dimensional array of the same total size and with the same total number of elements.

User Action

Verify that all subscript references to the array reference the correct array element.

COMMANDMACRO, Extraneous text "*text*" at the end of the command line macro "*macro*" is ignored.

Description

A command line macro define contains an invalid macro name. The compiler will define the macro name listed in the message.

User Action

Correct the command line invocation.

COMPILERBUG, Bug found in compiler: *bug*.

Description

This message indicates that the compiler detected a bug within itself.

User Action

Please report the compiler bug and include an example program that reproduces the problem.

COMPLEXEXT, The complex data type is a new feature in the C99 standard. Other C compilers may not support this extension.

Description

This is a new language feature in the C99 revision of the standard. While having a standard specification for portability, the feature may not yet be available in all of the compilers you use.

User Action

Determine whether or not the use of this feature will cause portability problems for this code.

COMPLEXNA, The complex data types are not supported in this language mode. This will be treated as an identifier in this compilation.

Description

Support for the complex data types is only available in certain language modes. Support is not present when the compiler is in VAX C, K & R (common), or strict ANSI89 standard modes. In these language modes _Complex and _Complex_I will be treated as identifiers.

User Action

Compile using one of the other compilation modes.

COMPLEXNA1, The complex data types are not supported on this platform. This will be treated as an identifier in this compilation.

Description

The complex data type is not supported on the VAX platform.

User Action

Remove use of the complex types or compile the application on a platform that does support the complex data types.

CONFLICTHINTS, *context*this hint value contridicts a related hint at *where*. The hints will be ignored.

Description

This program has supplied hints for either both branches of an if/else or both the second and third operand of a conditional operator. In these cases the two hint values must add to one.

User Action

Correct the hints.

CONLINKREG, Conflicting register usage between "*first set*" and "*second set*". Pragma is ignored.

Description

The same register was specified in two different register lists of a #pragma linkage directive. The compiler will ignore the entire pragma.

User Action

Correct the directive.

CONPSECTATTR, Conflicting psect attribute overrides previous attribute.

Description

A psect attribute specified in a #pragma extern_model directive contradicts an attribute specified earlier in the directive. This attribute will override the one specified earlier.

User Action

Remove one of the contradictory psect attributes.

CONSTCOMPLIT, *context*accepting a compound literal as a constant is a language extension. The compound literal will be treated as a cast expression.

Description

A compound literal appears in a context where a constant expression is required. The C standard does not list compound literals as a form of operand that is allowed in a constant expression, so using a compound literal in this context is not maximally portable. The compiler will treat the compound literal as if it were a cast expression, which is a form of operand that the standard lists as being allowed in constant expressions.

User Action

For maximum portability, replace the compound literal with a cast expression.

CONSTFOLDNS, *context*the libraries on this platform do not yet support compile-time evaluation of the constant expression "*expression*".

Description

Compile-time evaluation of constant expressions requires underlying support in the libraries available to the compiler at compile-time, and this expression contains an operator that is not yet implemented in those libraries.

User Action

If possible, replace part of the constant expression with a variable of the same value.

CONSTFUNC, Ignoring const type qualifier in declaration of *name*.

Description

The const type qualifier cannot be used with a function type. The compiler will ignore the type qualifier.

User Action

Remove the type qualifier.

CONSTINWRT, Const variable resides in wrt extern model.

Description

The current extern model places all external objects in a modifiable section. Placing an object with a const type qualifier in such a section means that there is no run-time protection against writing to the object. This might not have been what you intended.

User Action

Place const objects in sections that cannot be modified.

CONSTNOINIT, *context*the const object "*name*" is uninitialized.

Description

A defined or tentatively-defined const object has not been initialized. This would not be valid in C++. It is also considered good programming practice to initialize all const objects with their value.

User Action

Either remove the const type modifier, or supply an initializer for the object.

CONSTSTOCLS, *context*the const object "*name*" has no explicit storage class. In C, its storage class defaults to "extern"; in C++, it defaults to "static". Add an explicit "extern" or "static" keyword.

Description

One of the more signifcant and confusing differences between C and C++ is their treatment of file scope const objects declared without a storage class. C will give the object extern storage class, making the object visible in other compilation units. C++ will give the object static storage class. This can cause an undefined symbol error when other compilation units try to reference the symbol.

User Action

Add an explicit "extern" or "static" keyword to the declaration.

CONTFILE, A file ends with a continuation character.

Description

All source files, even those included via the #include preprocessing directive, must not end with a backslash continuation character.

User Action

Either remove the continuation character or add an additional line to the source program that does not end in a continuation character.

CONTROLASSIGN, *context*the assignment expression "*expression*" is used as the controlling expression of an if, while or for statement.

Description

A common user mistake is to accidentally use assignment operator "=" instead of the equality operator "==" in an expression that controls a transfer. For example saying if (a = b) instead of if (a == b). While using the assignment operator is valid, it is often not what was intended. When this message is enabled, the compiler will detect these cases at compile-time. This can often avoid long debugging sessions needed to find the bug in the user's program.

User Action

Make sure that the assignment operator is what is expected.

CONVARASLIT, *context* the use of the const variable "*name*" in place of a literal constant is a language extension.

Description

VSI C will allow a non-volatile const variable that has been initialized to be used in contexts where a constant is required. For example, as the bounds specifier to a file scope array. This is an extension to standard C. Other C compilers might not successfully compile a program that uses this extension.

User Action

Use the constant value instead of the variable.

CRXCOND, Common Data Dictionary description extraction condition.

Description

Something went wrong while trying to get the CDD record description from the CDD. The error message that follows gives more information about the nature of the problem.

User Action

If necessary, correct the indicated condition in the CDD record description or with the user environment.

CVIDXOVFL, module uses more than 65536 CodeView type indices

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

CVTDIFTYPES, *context*"*expression*" of type "*type*", is being converted to "*target type*".

Description

In certain modes, the compiler will allow assignments or comparisons between pointer and integer types. This is an extension to standard C. Other C compilers might not successfully compile a program that uses this extension.

User Action

Use a cast operator to convert one operand to the other.

CVTU32TO64, *context*an unsigned 32-bit integer constant that has its high-order bit set has been converted to a signed 64-bit type. The conversion will not sign-extend.

Description

This message indicates a conversion that may produce unexpected results on this platform because the destination type is a 64-bit type instead of a 32-bit type.

User Action

If this is the intended behavior, first cast the constant to an unsigned 64-bit type.

CXXCOMMENT, C++ style comments (//) may not be portable.

Description

C++ style comments have been detected on this line. Although they have been accepted by VSI C in this language mode, they will not be accepted by all compilers or by VSI C in strict C89 standard mode.

User Action

Replace C++ style line comments (//) with equivalent C comments (/* ... */) if portability is a concern.

CXXKEYWORD, "*C++ keyword*" is a keyword in C++. Using it as an identifier in your C program will prevent porting your program to C++.

Description

This identifier is a keyword in C++. The program is, therefore, not a valid C++ program.

User Action

Choose a different name for the identifier.

CXXPRAGMANA, The VSI C++ pragma "*pragma name*" is not supported by VSI C. The pragma will be ignored.

Description

The compiler has encountered a pragma that is supported by VSI C++ but is not supported by VSI C. The compiler will ignore the pragma.

User Action

Remove the pragma or compile the program with VSI C++.

DCLMISMATLNK, The declaration of "*name*" has *number* parameter(s) but its linkage "*name*" has *number*. Standard linkage will be used.

Description

The number of parameters specified in a declaration does not match the number of parameters specified by the special linkage associated with this function or typedef. The special linkage was specified via the #pragma use_linkage directive. Because of this mismatch, the compiler will ignore the special linkage and use the standard linkage instead.

User Action

Make sure the number of parameters specified by the special linkage match the number of parameters in the function.

DCLMISMATLNK0, The declaration of "*name*" has an unknown number of parameters and cannot be used with the linkage "*name*". Standard linkage will be used.

Description

If a special linkage specifies parameter information, the declaration must not specify an unknown or variable number of parameters. The special linkage was specified via the #pragma use_linkage directive. Because of this mismatch, the compiler will ignore the special linkage and use the standard linkage instead.

User Action

Make sure the number of parameters specified by the special linkage match the number of parameters in the function type.

DCLMISMATLNK1, *where* "*name*" *modifier* has a floating type but its linkage "*name*" specifies an integer register. Standard linkage will be used.

Description

A parameter or return value of a function type is a floating type, but the corresponding parameter or return value in the special linkage specifies an integer register. The special linkage was specified via the #pragma use_linkage directive. Because of this mismatch, the compiler will ignore the special linkage and use the standard linkage instead.

User Action

Make sure the register specified by the special linkage matches the type of of the corresponding parameter and return value of the function type.

DCLMISMATLNK2, *where* "*name*" *modifier* requires an integer register but its linkage "*name*" specifies a floating register. Standard linkage will be used.

Description

A parameter or return value of a function type is an integer type, but the corresponding parameter or return value in the special linkage specifies a floating register. The special linkage was specified via the #pragma use_linkage directive. Because of this mismatch, the compiler will ignore the special linkage and use the standard linkage instead.

User Action

Make sure the register specified by the special linkage matches the type of of the corresponding parameter and return value of the function type.

DCLMISMATLNK3, *where* "*name*" has a size that is incompatible with the number of registers specified by its linkage "*name*". Standard linkage will be used.

Description

The size of a parameter or return value of a function type is incompatible with the size specified by the special linkage. The special linkage was specified via the #pragma use_linkage directive. Because of this mismatch, the compiler will ignore the special linkage and use the standard linkage instead.

User Action

Make sure the number of registers specified by the special linkage match the type of the corresponding parameter and return value.

DCLMISMATLNK4, *where* "*name*" *modifier* has a type that is not allowed because the it has the linkage "*name*". Standard linkage will be used.

Description

Using a special linkage places certain restrictions on the type of a function's parameters and return value. In general, the type must be a scalar type that can be represented by a register or registers on this platform. In cases where some other type is used, the compiler will ignore the special linkage and use the standard linkage instead.

User Action

Either remove the name from the #pragma use_linkage directive that specified the special linkage, or modify the type to be acceptable to the special linkage.

DCLMISMATLNK5, "*name*" has a void return type but its linkage "*name*" specifies a return location. Standard linkage will be used.

Description

If a special linkage specifies return value information, the declaration must not specify a void return type. The special linkage was specified via the #pragma use_linkage directive. Because of this mismatch, the compiler will ignore the special linkage and use the standard linkage instead.

User Action

Make sure the return value specified by the special linkage matches the return type.

DCLMISMATLNK6, *where* "*name*" *modifier* has float _Complex or double _Complex type. The corresponding floating point registers in linkage "*name*" must be consecutive. Standard linkage will be used.

Description

Using a special linkage places certain restrictions on the type of a function's parameters and return value. Whenever float _Complex or double _Complex types are used, they linkage must specify two consecutive floating point registers. The compiler will ignore the special linkage and use the standard linkage instead.

User Action

Either remove the name from the #pragma use_linkage directive that specified the special linkage, or modify the linkage to use consecutive floating point registers.

DECCONSTLARGE, Decimal constant value too large.

Description

A decimal constant used in a preprocessor directive is too large. The value of the constant will be undefined.

User Action

Decrease the value of the constant.

DECLAFTERSTMT, Placing a declaration after a statement is a new feature in the C99 standard. Other C compilers may not support this feature.

Description

This is a new language feature in the C99 revision of the standard. While having a standard specification for portability, the feature may not yet be available in all of the compilers you use.

User Action

Determine whether or not the use of this feature will cause portability problems for this code.

DECLARATOR, Invalid declarator.

Description

A declaration did not contain an identifier that specifies the item to be declared.

User Action

Specify a declarator in the declaration.

DECLINFOR, Placing a declaration in a for loop is a new feature in the C99 standard. Other C compilers may not support this extension.

Description

This is a new language feature in the C99 revision of the standard. While having a standard specification for portability, the feature may not yet be available in all of the compilers you use.

User Action

Determine whether or not the use of this feature will cause portability problems for this code.

DECLSPECEXT, __declspec is a language extension.

Description

The __declspec storage class modifier is a language extension of VSI C. Other C compilers might not successfully compile a program that uses the extension.

User Action

Be aware of this extension if you wish to port the code.

DEFINOTHER, Another file in this compilation contains an external definition of a function named "*name*", or declares it as a variable with external linkage, at *where*.

Description

In a compilation where interfile optimization has been selected (-ifo on UNIX, /PLUS_LIST_OPTIMIZE on OpenVMS), the compiler has detected more than one definition of a function using the same external name, or has found that a function and a variable have the same external name. An external function can have only a single definition. And a given identifier with external linkage can refer either to a function or to a variable, but not both.

User Action

Remove or rename one of the names.

DEFINOTHER1, The external variable "*name*" was defined as an external function in another module of this compilation at *where*.

Description

In a compilation where interfile optimization has been selected (-ifo on UNIX, /PLUS_LIST_OPTIMIZE on OpenVMS), the compiler has detected a name with external linkage defined as a variable in one compilation unit and a function in another.

User Action

Remove or rename one of the definitions.

DEFINOTHER2, This declaration of "*name*" specifies a different type than the declaration in another module of this compilation at *where*.

Description

In a compilation where interfile optimization has been selected (-ifo on UNIX, /PLUS_LIST_OPTIMIZE on OpenVMS), the compiler has detected a name with external linkage declared with different types in two different modules. Although the runtime behavior may be as intended and match the behavior when the modules are separately compiled without interfile optimization, the behavior is not well defined unless the types are compatible.

User Action

Modify one or more of the declarations to make the types compatible.

DEFINOTHER3, This declaration of "*name*" specifies a different thread-local attribute than a declaration in another module of this compilation at *where*.

Description

In a compilation where interfile optimization has been selected (-ifo on UNIX, /PLUS_LIST_OPTIMIZE on OpenVMS), the compiler has detected a name with external linkage declared thread-local in one module and not thread-local in another. This can lead to unexpected results at runtime.

User Action

Modify one the declarations to make the thread-local attributes match.

DEFPARMTYPE, There is no declaration for the old-style function parameter "*name*". Type defaulted to int. This is a violation of the C99 standard.

Description

The parameter of an old-style function definition was not declared. It will default to int type. Omitting the type specifier is not valid in C99, and is often considered poor programming practice.

User Action

Declare the parameter. VSI also recommends that old-style function definitions be replaced by prototype-format definitions.

DEFRETURNTYPE, The type of the function *name* defaults to "int".

Description

A function definition did not include a type specifier for the function's return value. It will default to int. This might not be what you intend. This is also a violation of the C99 Standard.

User Action

It is a good programming practice to give all function definitions explicit return types.

DESIGBADARR, *context*, a struct/union designator cannot be used with an object of array type.

Description

An initialization designator must match the type of the object being initialized. In this initialization, the current object is an array so a struct/union designator is not allowed.

User Action

Correct the initialization.

DESIGBADCOMP, *context*, an array designator cannot be used with an object of struct or union type.

Description

An initialization designator must match the type of the object being initialized. In this initialization, the current object is a struct or union, so an array designator is not allowed.

User Action

Correct the initialization.

DESIGBADIND, *context*, the constant expression "*expression*" in an array element designator is not a positive integer.

Description

An array-element designator must be an constant expression that yields a positive integer value.

User Action

Correct the element designator.

DESIGBADIND1, *context*, the array element designator "[*expression*]" specifies an element beyond the end of the array.

Description

An array element designator must specify a valid array element.

User Action

Correct the element designator.

DESIGNATIONNA, The use of a designation in an initializer list is not supported in this compilation mode.

Description

Initializer lists that contain designations are a new feature in the C99 revision of the C standard. VSI C will only support this extension in relaxed mode and strict c99 mode.

User Action

Use a compilation mode that supports the use of designations.

DESIGNATORUSE, The use of a designation in an initializer list is a new feature in the C99 standard.

Description

Initializer lists that contain designations are a new feature in the C99 revision of the C standard. Other compilers may not support this feature.

User Action

Be aware of this portablility issue.

DESIGNOMEMB, *context*, the component designator "*name*" is not a member of the current structure or union object being initialized.

Description

An initialization designator specifies a struct or union member that is not a member of the current struct or union object.

User Action

Correct the initialization.

DESIGSCALAR, *context*, a designator cannot be used with an object of scalar type.

Description

An initialization designator can only be used on objects of array, structure, or union type. In this initialization, the current object being initialized is a scalar type so a designator is not allowed.

User Action

Correct the initialization.

DIFFEXMODEL, This redeclaration of "*name*" specifies a different extern model than a previous declaration of the variable at *location*.

Description

Two declarations of the same variable use different extern models. The extern model is specified by a #pragma extern_model directive that appears before the declaration in the source. This redeclaration may cause unexpected behavior.

User Action

All declarations of a variable should use the same extern model.

DIFFTYPEQUALS, *context*the type of "*name*" has different type qualifiers than the previous declaration at *location*. The resulting type will be the composite of the two types.

Description

The C standard permits redeclaration and formation of a composite type only when the two types being considered are compatible, and types with different type qualifiers are not compatible. VSI C allows this redeclaration for consistency with some other C compilers, and will form a composite type with all of the type qualifiers from both declarations. Be aware that these declarations may not be accepted by other C compilers.

User Action

Modify the declarations so that they use identically qualified types.

DIRECTVNOCPP, "*Directive text*" is not recognized as a preprocessing directive in nopreprocessing mode, and is being ignored.

Description

An invalid preprocessing directive was encountered in a compilation performed with the -nocpp option. When using the -nocpp option, only a limited number of preprocessing directives, such as #pragma and #line, can appear in the program. The compiler will ignore the rest of the line.

User Action

Either remove the directive or compile without the -nocpp option.

DISREDECL, *context*the type of the external "*name*" is not compatible with the type of a declaration of "*name*" in another name scope at *location*.

Description

The same external identifier has been declared in different scopes with incompatible types. This might not have been what you intended.

User Action

Change all declarations of the same external identifier to use the same type.

DOLLARID, Extension: A '$' was encountered in an identifier.

Description

Accepting a "$" character in an identifier is an extension of VSI C. The program might not compile with other C compilers.

User Action

Be aware of this if you wish to port the program.

DONOTAPPLY, linkage, assert or hint information for built-in function *name* is ignored.

Description

A built-in function is always handled specially. There is no actual function call to which linkage, assert or hint information could be applied.

User Action

Remove the name of the built-in function from this pragma.

DUPCASE, The switch statement containing this case label already has a case label for "*number*".

Description

A switch statement contains more than one case label for the same case value.

User Action

Remove the duplicate case label.

DUPDEFAULT, The switch statement containing this default label already has a default label.

Description

A switch statement can contain only one default label.

User Action

Remove the duplicate default label.

DUPENUM, *context*the enumerator "*name*" is not unique.

Description

An enumerator constant is declared more than once with the same value. While this is accepted by VSI C, it is not allowed by the C standard.

User Action

Either use a different enumerator name or remove the previous declaration of the name.

DUPEXTERN, The declaration of "*name1*" will map to the same external name as the declaration of "*name2*" at *where*.

Description

The compiler has detected a case where two different names in a program will map to the same external name in the output object file. This can cause unpredictable results at runtime. This will most often happen when the /NAMES=UPPERCASE or /NAMES=LOWERCASE qualifier causes two names with different case spellings to map to the same external name.

User Action

Either use the /NAMES=AS_IS qualifier, or modify one of the names.

DUPLABEL, The label "*name*" is already defined in this procedure at *location*.

Description

A label has already been defined. Each function can define each label only once.

User Action

Remove the duplicate label definition.

DUPLINK, Duplicate linkage pragmas for linkage name "*linkage name*".

Description

The same linkage specifier has been defined in more than one #pragma linkage directive.

User Action

Declare each linkage only once.

DUPLPRAGASS, #pragma assert directive specified for the function name *name* while different #pragma assert was specified for its type.

Description

Duplicate assertion can't be specified for a function. Check whether #pragma assert was mistakenly specified for the same function more than once, or function's type is declared in a typedef which in turn has its own #pragma assert directive.

User Action

Either remove duplicate #pragma assert directive, or change assertions, or fix spelling of the function name or typedef.

DUPPARM, *context*"*name*" is a duplicate parameter name.

Description

The parameter identifier list of an old-style function definition uses the same identifier more than once.

User Action

Each identifier in the parameter list must be unique. VSI also recommends that old-style function definitions be replaced by prototype-format definitions.

DUPSTATIC, There is a redundant use of the keyword "static" in this array declaration.

Description

In C99 the keyword "static" may appear at most once in the outermost array-bounds specifier of a function parameter in a function prototype.

User Action

Remove redundant occurrences(s) of "static" from the array declaration

DUPSTORCLS, *context*the same storage class modifier occurs more than once.

Description

This declaration specifies the same storage class modifier more than once.

User Action

Remove the extra uses of the storage class modifier.

DUPTYPEDEF, *context*"*name*" has a duplicate typedef at *where*. This might not be portable.

Description

The same typedef has been declared to the same type more than once. Standard C does not allow this and other compilers might not accept it.

User Action

Remove the redundant declaration.

DUPTYPESPEC, *context*the same type specifier occurs more than once.

Description

The same type specifier appears more than once in the same declaration. The redundant specifier will be ignored.

User Action

Remove the duplicate type specifier.

DUPTYPQUAL, *context*there is a redundant use of type qualifier "*const or volatile*".

Description

The same type qualifier appears more than once in a type specifier. This violates the C89 standard. Other compilers may not accept this program. Note that C99 will allow redundant qualifiers.

User Action

Remove the redundant type qualifier.

ELIFIGNORED, Out of place #elif directive ignored.

Description

An #elif preprocessing directive was encountered outside of an #if/#endif body. The directive will be ignored.

User Action

Remove the directive.

ELLIPSEARG, Standard C does not permit the use of an ellipsis as an only argument.

Description

Standard C requires at least one formal parameter be declared before the ellipses. This declaration might not be portable to other C compilers.

User Action

Recode the function declaration to contain at least one formal parameter.

ELLIPSEPARM, *context*a parameter with type "*type*" matches an ellipsis in previous declaration at *location*.

Description

A function that has been previously declared as taking variable arguments is now redeclared as using a different number of formal parameters before the start of the variable argument list. This redeclaration might not be portable to other C compilers.

User Action

Recode the function declarations to match each other.

ELLIPSISEND, No tokens may follow ... in a formal parameter list.

Description

The ellipsis may only appear at the end of a formal parameter list. Everything after that is being ignored.

User Action

Remove the unexpected token.

ELSEIGNORED, Out of place #else directive ignored.

Description

An #else preprocessing directive was encountered outside of an #if/#endif body. The directive will be ignored.

User Action

Remove the directive.

EMBEDCOMMENT, A comment is neither preceded nor followed by white space.

Description

A comment is neither preceded nor followed by white space. In certain modes the compiler will paste the tokens before and after the comment together to form a single token. This behavior is not valid in standard C. Writing programs that rely on this behavior might prevent the program from being compiled on other platforms.

User Action

Add white space before or after the comment, or use the ## operator to paste tokens together.

EMPTYCHARCONST, Empty character constant.

Description

In some modes the VSI C compiler will allow a null character constant. The compiler will give this constant a value of zero. Accepting an empty character constant is a language extension. Empty character constants are not valid in standard C. Writing programs that rely on this behavior might prevent the program from being compiled on other platforms.

User Action

Replace the empty character constant with '\0'.

EMPTYFILE, Source file does not contain any declarations.

Description

This source file contains no declarations. This might not have been what you intended. For example, perhaps a necessary macro was not defined.

User Action

Every source program should contain at least one declaration.

EMPTYINIT, An initializer list without an expression is not valid. The compiler will replace the empty expression with the constant 0.

Description

The C standard requires that an initializer list contain an expression. The compiler has encountered one without an expression. The compiler will treat the empty list ({}) as if it contained a single zero ({0}). This is for compatibility with some other C compilers. Be aware that this syntax may not be accepted by other C compilers.

User Action

Supply an expression to the initializer.

EMPTYOBJ, Empty object file due to errors.

Description

An earlier condition will cause an empty object module to be created.

User Action

Correct the condition that was reported earlier.

EMPTYSTRUCT, Allowing struct/union type with no members is a language extension.

Description

The C standard requires that a struct/union type have at least one member. The VSI C compiler will accept this for compatibility with older compilers. The struct/union type will be treated as if it were declared { : 0; }

User Action

Provide at least one member for the struct/union.

ENUM16BIT, *context*the enumeration constant *name* is out of the range -32768 to 32767. This might not be portable.

Description

An enum constant is larger than can be represented in 16 bits. This would not be portable to a system with an int size of 16 bits.

User Action

Be aware of this if you wish to port to a system with an int size of 16 bits.

ENUMCALC, *context*the enum variable "*expression*" is used in an arithmetic operation.

Description

An enumerated type variable was used in an arithmetic operation. While this is valid in C, it might not have been what you intended.

User Action

Verify the use of the enum variable.

ENUMINIT, *context*the enumerator "*name*" is initialized to the nonintegral value "*expression*".

Description

An enum declaration contains an enumeration constant initializer that does not have an integer type. The initializer for an enumeration constant must be an integral constant expression.

User Action

Correct the initializer.

ENUMRANGE, *context*the enumeration constant "*name*" is out of range INT_MIN to INT_MAX and will be truncated.

Description

An enumeration constant must be representable as an int type. The specified value is outside the range of an int. In modes where this is a warning, the compiler will use the low-order bits to form the int value.

User Action

Use a valid constant value.

ENUMSANDINT, *context*allowing an enumeration type and a signed int to be compatible may not be portable.

Description

The standard states that enumeration types shall be compatible with an integer type. VSI C, along with most other C compilers, has chosen the signed int type to be compatible with enumeration types. Other compilers may chose another type such as unsigned int (the C standard even allows an implementation to choose different integer types depending on the values of the enumeration constants defined for the type). Therefore this program may not be accepted by other C compilers.

User Action

Insert a cast to make the types the same.

ENUMSNOTCOMPAT, *context*allowing two different enumeration types to be compatible is a language extension.

Description

The VSI C compiler allows two objects of different enumeration types to be compatible. The C standard specifies that enumeration types are distinct types. Therefore this program is not standard compliant and other C compilers may not accept it.

User Action

Use the same enumeration type or cast one type to the other.

ENUMUSED, *context*the enumerator name "*name*" has been used previously.

Description

The specified enumerator name has been previously declared as something other than an enumerator.

User Action

Either use a different enumerator name or remove the previous declaration of the name.

ENVIRSTKDIRTY, At the end of the compilation the pragma *name* stack was not empty. This may indicate a coding error.

Description

The program being compiled has saved the named pragma state more often than it has restored it. Good coding practice calls for the pragma state to be restored some point after it has been saved. This condition may indicate the accidental failure to restore the state.

User Action

Make sure each pragma save has a corresponding pragma restore.

ERRORLIM, diagnostic message limit exceeded

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

ERRORMESSAGE, #error*errormsg*

Description

An #error directive was encountered. This message will include the text that follows the directive in the source program.

User Action

Remove the #error directive, or supply the proper macro definitions so that the compiler will skip the directive.

ESCOVERFL, Invalid escape sequence encountered.

Description

An escape sequence in a character or string literal specifies a value outside the range of a character or wide character.

User Action

Specify a valid escape sequence.

EXPANDEDDEFINED, Macro expansion includes the token "defined", which will be treated as an operator. This might not be portable.

Description

A macro expanded during the processing of a preprocessor #if directive included the token "defined". The VSI C compiler will treat this as the defined preprocessing operator. Other compilers might treat this differently.

User Action

Rewrite the macro not to use the "defined" operator.

EXPNOTRES, expression does not contribute to result

Description

The compiler has detected a source expression that does not contribute to the result. This may not be what you expected.

User Action

Verify the expression is what you intend.

EXPRCVTINT, The expression "*expression*" has been converted to integer.

Description

In certain modes, VSI C will allow switch expressions or case constants to be non-integer types. The expression or constant will be converted to int. In one of these cases, this warning will be issued.

User Action

Cast the switch expression to an integer type or use an integer case constant.

EXPRNOTINT, The expression "*expression*" has *type* type, which is not integral.

Description

An expression that is required to have an integer type had a type that is not integral. This is not valid. An example of a situation where an integer is required is that in most modes VSI C requires that the switch control expression have integer type.

User Action

Modify or cast the expression so that it has integer type.

EXPRNOTUSED, *context*the expression "*expr*" is never used.

Description

The compiler has detected an expression that is not used, and might not have a side-effect. This might not have been what you intended.

User Action

If the expression has a desired side-effect, the message can be ignored. Otherwise, you might want to consider removing the expression.

EXTENDTYPE, This platform specific type is a language extension.

Description

The use of the types __int8, __int16, __int32, __int64, or other type specifiers beginning with leading double underscores might not be portable to other platforms or to other C compilers.

User Action

Be aware of this portability concern.

EXTERNINIT, VSI C allows the initialization of a variable with extern storage class. This differs from the VAX C behavior.

Description

VAX C does not allow a variable with extern storage class to be initialized. VSI C will allow this, even in vaxc mode.

User Action

Be aware of this difference if you plan to compile the source with VAX C.

EXTERNPOP, This "restore" has underflowed the extern model's stack. No corresponding "save" was found.

Description

The extern_model stack, managed by the #pragma extern_model and #pragma environment directives, contains more restores than saves. This could signify a coding or logic error in the program.

User Action

Make sure each restore has a corresponding save.

EXTPREAFTER, This directive will not set the extern_prefix of "*name*" because there is a previous declaration of the identifer with external linkage at *where*.

Description

When an identifier is specified in a #pragma extern_prefix, the declaration of that identifier must appear after the #pragma.

User Action

Reorder the declaration and the #pragma so that the #pragma comes first.

EXTPREAGAIN, This directive overrides the extern_prefix for "*name*" specified by an earlier #pragma extern_prefix at *where*.

Description

Two #pragma extern_prefix directives have specified different non-empty extern_prefixes for the same identifier. In such cases the later directive will set the extern_prefix for the identifier.

User Action

If it is necessary to respecify the extern_prefix for an identifier, first remove the prefix (by setting it to an empty string) and then specify the new prefix in a subsequent #pragma.

EXTPRENODECL, There is no identifier named "*name*" with external linkage declared in this compilation unit.

Description

A #pragma extern_prefix directive specifies an extern prefix for an identifier that is not declared with external linkage in the compilation unit. This may not have been what you intented.

User Action

Remove the identifier from the #pragma extern_prefix, or declare it with external linkage, or set the prefix for this identifier to an empty string.

EXTRABRACES, *context*, the value is enclosed within too many pairs of braces.

Description

An initializer contains too many open braces for the object being initialized.

User Action

Reduce the number of braces.

EXTRAMODULE, Redundant "#pragma module" or "#module" directive ignored.

Description

A compilation unit can contain only one #pragma module or #module directive. All subsequent directives will be ignored.

User Action

Remove the extra directives.

EXTRAPRAGARGS, Extra pragma arguments to #pragma *pragma* were found. Pragma is ignored.

Description

Unexpected arguments were found at the end of a #pragma directive. The directive will be ignored.

User Action

Remove the extra arguments.

EXTRASEMI, Extraneous semicolon.

Description

An extra semicolon was found at the end of a declaration. It will be ignored.

User Action

Remove the extra semicolon.

FALLOFFEND, The last statement in non-void function "*name*" is not a return statement.

Description

A function that returns a value does not end with a return statement. If function execution reaches the end of the function, the implied return statement that executes will return an undefined value. This might not have been what you intended.

User Action

End the function with a return statement that specifies a return value.

FBFILENOTFOUND, Feedback file not found: *text*

Description

The specified feedback file could not be found by the compiler.

User Action

Specify the correct file name.

FILECLOSE, An error occurred while attempting to close a source file: *problem*.

Description

An unexpected error occurred while closing a source file. The message text will contain additional information about the failure.

User Action

Correct the condition that caused the failure.

FILENOTFOUND, File not found: *text*

Description

The specified file could not be found by the compiler.

User Action

Specify the correct file name.

FILEREAD, An error occurred while attempting to read a source file: *problem*.

Description

An unexpected error occurred while reading a source file. The message text will contain additional information about the failure.

User Action

Correct the condition that caused the failure.

FILESCOPEVLA, *context*the file-scope identifier "*name*" cannot be declared with a variably modified type.

Description

Only ordinary identifiers with block scope and without storage class extern, or ordinary identifiers with function prototype scope can be declared with a variably modified type.

User Action

Correct the declaration.

FINBRANCH, A goto to the label "*label*" branches into a finally handler.

Description

A goto statement tried to transfer into a finally handler. This is illegal.

User Action

Modify the goto or move the label outside the handler.

FLEXARRAYELEM, *context*allowing an array element to be a struct with a flexible array member is a language extension.

Description

The C99 standard allows the final element of a struct with more than one named member to have incomplete array type. Such a member is called a flexible array member. The standard does not allow such a struct (and any union containing, possibly recursively, a member that is such a struct) to be an array element. Other C compilers may not support this extension.

User Action

Be aware of this extension if you wish to port the code.

FLEXARRAYMEM, *context*allowing the struct member, "*name*" to be a struct with a flexible array member is a language extension.

Description

The C99 standard allows the final element of a struct with more than one named member to have incomplete array type. Such a member is called a flexible array member. The standard does not allow such a struct (and any union containing, possibly recursively, a member that is such a struct) to be a member of another structure. Other C compilers may not support this extension.

User Action

Be aware of this extension if you wish to port the code.

FLOATCONSQUAL, The *float_const_qual* is not valid in strict ANSI mode and will be ignored.

Description

The -float_const option cannot be used in strict ANSI mode. The option will be ignored.

User Action

Either remove the -float_const option or use a different mode.

FLOATCONST, Ill-formed floating constant.

Description

An invalid floating constant was encountered.

User Action

Correct the floating constant.

FLOATERR, *context*a floating point error occurs in evaluating the expression "*expression*".

Description

A floating-point error occurred while evaluating a constant expression. This is often caused by an invalid floating-point number. The value of the expression is undefined.

User Action

Correct the floating-point constant expression.

FLOATOVERFL, *context*floating-point overflow occurs in evaluating the expression "*expression*".

Description

A floating-point overflow occurred while evaluating a constant expression. The value of the expression is undefined.

User Action

Correct the floating-point constant expression.

FLOATTOINT, *context*"*expr*" is being converted from *type* type to int type.

Description

The C language requires that this expression be of integer type. In most cases the compiler will emit an error for this case. In VAX C mode, the compiler emits this warning and converts the expression to int type. This matches the behavior of VAX C.

User Action

If the VAX C behavior is what you intended, cast the expression to int to silence the diagnostic. Otherwise, recode the expression to reflect your intent.

FMTNOTSTR, argument *number* of this function is not of type char * but corresponds to the format string specified by the #pragma assert directive at *location*. The format func_attr will be ignored.

Description

The format attribute causes the format string to be checked if it is a string constant. The format parameter can't be a format string because it is not declared as a char * type. The format attribute will be ignored.

User Action

Either remove the format assertion from the directive, correct the position of the format argument in the assertion, or declare the format argument as a "char *" in the proper position in the function prototype.

FNAMETOOLONG, The file name "*name*" in this directive is too long.

Description

A preprocessing directive has specified a file name that is too long for this platform.

User Action

Supply a valid file name

FORMATATTR, *context*the arguments to *function name* do not match the assertions of its format attribute. The format argument or the argument preceeding the first argument to check is missing.

Description

The format attribute of this function asserts that the format argument exists and will be checked if it is a string constant. The first argument to check, if non-zero, identifies the argument corresponding to the ellipsis in the function declaration and asserts that the argument preceeding it exists.

User Action

Modify either the function call or the format attribute so that they match.

FOUNDCR, A carriage-return character was encountered; it is being treated as white space.

Description

The compiler encountered a carriage-return character some place other than inside a character or string constant. The compiler will treat the carriage-return as white space.

User Action

The source might have been created by some non-standard means. If possible, replace all carriage-return characters outside of character or string constants with white space.

FREGNEEDSIEEE, Use of the floating register "*regnum*" in a #pragma linkage directive requires the /FLOAT=IEEE_FLOAT qualifier.

Description

On IA-64, VAX floating-point data is passed in general registers. VSI C requires that any program that uses a floating point register in a linkage directive must be compiled with IEEE floating-point.

User Action

Compile with IEEE floating-point. Another option would be to remove the floating point registers from the linkage.

FUNCELEMENT, *context*the element type of an array type is a function type.

Description

The compiler has encountered an array with an element type of function. An array element must be an object type.

User Action

Change the type of the array element.

FUNCIDLIS, *context*the identifier "*id*" is not the name of a type. All parameter information in this declaration will be ignored.

Description

The declaration is most likely a malformed prototype-style function declaration. In a prototype-style declaration, each parameter must have a type. The identifier named in the message might be intended to be the (optional) name of a formal parameter and the type specification was mistakenly omitted, or it might be intended to be the name of a type but no typedef declaration for it is visible. Alternatively, the declaration might be intended to correspond to an old-style function definition, and mistakenly contains a formal parameter name in the declaration. Old-style function definitions list the names of formal parameters (without types) inside the parentheses, but old-style function declarations contain nothing inside the parentheses.

User Action

Correct the declaration.

FUNCINIT, The declaration of the function "*name*" includes an initializer.

Description

A function declaration cannot contain an initializer.

User Action

Remove the initializer from the declaration.

FUNCMEM, The member *name* has a function type.

Description

A struct or union member is declared with function type. This is not valid.

User Action

Correct the member declaration.

FUNCMIXPTR, *context*function types differ because this declaration specifies "*type1*" and a previous declaration specifies "*type2*".

Description

A function redeclaration differs from an earlier declaration of the same function because the pointer size of one of the arguments or the return result is different.

User Action

Use the same pointer size for all declarations of the function.

FUNCNOTDEF, The function "*name*" has non-extern storage class, occurs in a context that requires its definition, and has no definition. The storage class has been changed to extern.

Description

In certain modes, the compiler will allow a static function to be declared within the scope of another function. If this function is referenced, then it must also be defined in the compilation unit. If the function is not defined, this message will be output, and the earlier static declaration will be changed to extern.

User Action

Define the static function with compilation unit.

FUNCNOTFUNC, In this function definition, "*name*" has *type* type instead of a function type.

Description

A function definition does not have a function type. This can occur if the definition did not contain an open/close parenthesis pair.

User Action

Change the definition to specify a function type.

FUNCREDECL, *context*function types differ because one has no argument information and the other has an ellipsis.

Description

Two function types, used in an operation or a redeclaration of a function, are different because one uses ellipses and the other does not. Older compilers will accept this, but it is not valid standard C.

User Action

If used in an operation, a cast should be inserted. If used in a redeclaration, the redeclaration should be removed or modified.

FUNCSTORCLS, *context*a function has an explicit storage class other than "static" or "extern".

Description

This declaration specifies a storage class that is not valid for a function. If an explicit storage class is used in a function declaration, it must be either static or extern.

User Action

Either remove the storage class specifier, or use one of the valid storage classes.

FUNCSTORMOD, *context*a function cannot have this storage class modifier. Modifier ignored.

Description

A function cannot be declared with this storage class modifier. The only valid storage class modifier for a function declaration is __inline. The modifier is ignored by the compiler.

User Action

Remove the storage class modifier from the function declaration.

FUNCSTRCLS, The block-level declaration of the function "*name*" specifies an explicit storage class other than extern.

Description

A block-level declaration of a function has specified an explicit storage class other than extern. VSI C will change the storage class to extern.

User Action

Either remove the storage-class specifier, or change it to extern.

FUTUREKEYWD2, "inline" is a keyword in the C99 revision of the C standard. Using it as an identifier will prevent your program from conforming to that standard.

Description

The token inline has been selected as a keyword in the C99 release of the C standard. Because the program uses it as an identifier, the program will not conform to that standard.

User Action

Change the name of the identifier.

FUTUREKEYWORD, "restrict" is a keyword in the C99 revision of the C standard. Using it as an identifier will prevent your program from conforming to that standard.

Description

The token restrict has been selected as a keyword in the C99 release of the C standard. Because the program uses it as an identifier, the program will not conform to that standard.

User Action

Change the name of the identifier.

GBLOUTSIDEINT, *context* the globalvalue constant *value* is outside the range of type int. This may cause unexpected results.

Description

The C compiler does not support globalvalue constants larger than int. The compiler preserves only the low-order 32 bits of the value, which will be sign-extended by the linker if the symbol is used in a certain contexts requiring a 64-bit value. This may cause unexpected results.

User Action

Use constants within the range of type int to initialize globalvalues, or use more portable constructs such as macro definitions or global const-qualified variables to share constant values among compilation units.

GBLREFINIT, The declaration of "*name*" specifies the globalref storage class and includes an initializer.

Description

A declaration with storage class globalref cannot include an initializer.

User Action

Either remove the initializer or use a storage class that will allow an initializer.

GCCINLINE, The inline and __inline keywords will be interpreted with GCC style semantics. To get C99 semantics, please specify -accept nogccinline.

Description

The C99 standard has a slightly different interpretation of the keyword inline than in GCC. The GCC __inline keyword also differs from the VSI C __inline keyword.

User Action

Use the command line specifier -accept nogccinline.

GEMARGSIZE, *context*the size of "*expression*" exceeds the implementation's limit of 2147483647 bytes on the size of a function argument.

Description

The size of a function argument exceeds the VSI C implementation limit.

User Action

Either reduce the size of the argument or consider passing it by reference.

GLOBALEXT, A storage class of globaldef, globalref, or globalvalue is a language extension.

Description

These storage classes are language extensions of VSI C. Other C compilers might not successfully compile a program that uses the extension.

User Action

These storage classes can be recoded using the more portable #pragma extern model.

```
globaldef int var1; globalref int var2; globalvalue int var3;
```

Can be written as:

```
#pragma extern_model save
#pragma extern_model strict_refdef
int var1;
extern int var2;
#pragma extern_model globalvalue
extern int var3;
#pragma extern_model restore
```

For more information, consult the #pragma extern_model documentation.

GOTSZOVFL, GOT table overflow for module *text*

Description

The object file required for this module is too complex.

User Action

Break the source program into several pieces so the individual objects will be simpler.

HEXOCTSIGN, In VAX C mode, the compiler will give this constant a signed type for compatibility with VAX C. This differs from the behavior specified in the C standard, which would give this constant an unsigned type.

Description

The C standard specifies that an octal or hexadecimal integer constant has an unsigned type when its value cannot be represented in a signed integer type, but can be represented in the corresponding unsigned integer type. Some older compilers, such as VAX C, will treat this constant as having a signed type. In VAX C mode, the compiler matches the behavior of VAX C. In other modes the compiler matches the behavior specified in the standard.

User Action

Be aware that this difference may cause porting problems if this program is compiled in a mode other than VAX C mode, or with a compiler that does not support this old behavior.

HEXOCTUNSIGN, The VSI C compiler conforms to the C standard and will give this constant an unsigned type. Some older compilers may give this constant a signed type.

Description

The C standard specifies that an octal or hexadecimal integer constant has an unsigned type when its value cannot be represented in a signed integer type, but can be represented in the corresponding unsigned integer type. Some older compilers will treat this constant as having a signed type.

User Action

Be aware of this difference if you plan to port this source to an older compiler.

HINTNOTFUNC, The identifier "*ident*" is not a declared function. It will be ignored in this #pragma hint func_attrs list.

Description

The identifiers in a #pragma hint func_attrs must be declared functions.

User Action

Either declare the function prior to the pragma or remove the identifier from the pragma.

HINTTOOBIG, *context*this hint value must not be greater than one. The hint will be ignored.

Description

This #pragma hint directives must take positive floating point values which is not greater than one.

User Action

Correct the hint.

IDEXPECTED, Identifier expected but not found.

Description

The compiler was expecting an identifier, but one was not found.

User Action

Correct the program syntax.

IDINPARENSEXT, *context*accepting an identifier enclosed in parentheses as the second argument to va_start is a language extension.

Description

The C standard states that the second argument to va_start must be an identifier. For compatibility with other C compilers, VSI C will accept an identifier enclosed in parentheses. Be aware that this program does not conform to the standard and may be rejected by other compilers.

User Action

Remove the parentheses.

IDPACKPOPPRAG, The identifier *name* from the pragma pack pop directive was not found on the top of the pragma pack stack.

Description

The identifier specified in the #pragma pack (pop, <identifier>) directive was not found on the top of the pragma pack stack. A previous #pragma pack pop or #pragma member_alignment restore may have already popped this identifier off the stack, the identifier may not have been previously pushed onto the stack, or extra elements are pushed on the stack on the top of element with the identifier, or the identifer may be spelled incorrectly.

User Action

Check the spelling of the identifier. Verify that the identifier was previously pushed onto the pack stack and not popped off by another #pragma pack pop or #pragma member_alignment restore, and all elements pushed on the top of the identifier are popped. Correct the directive(s).

IEEEASSUMED, Use of /ROUNDING_MODE qualifier implies /FLOAT=IEEE. Compilation will be performed as if /FLOAT=IEEE were specified on the command line.

Description

This compilation has specified an IEEE floating-point rounding mode without specifying /FLOAT=IEEE on the command line. The compiler will set the floating-point type to IEEE floating.

User Action

Specify /FLOAT=IEEE on the command line.

IEEEASSUMED1, Use of /IEEE_MODE qualifier implies /FLOAT=IEEE. Compilation will be performed as if /FLOAT=IEEE were specified on the command line.

Description

This compilation has specified an IEEE floating-point mode without specifying /FLOAT=IEEE on the command line. The compiler will set the floating-point type to IEEE floating.

User Action

Specify /FLOAT=IEEE on the command line.

IGNORECALLVAL, *context*the value returned from the function "*expression*" is not used - if this is intended, it should be cast to "void".

Description

A function that returns a value has been invoked, yet the value was not used. This might not have been what you intended.

User Action

Cast the function to void to suppress the message.

IGNOREEXTRA, Spurious token(s) ignored on preprocessor directive line.

Description

A preprocessing directive was supplied more arguments than it expects. The extra arguments will be ignored.

User Action

Remove the extra arguments.

IGNORETAG, *context*the tag "*name*" is redeclared, but will be ignored.

Description

The "struct" or "union" before the tag used in this declaration does not match that in the declaration of the tag. The "struct" or "union" at the earlier declaration of the tag will be used in this declaration.

User Action

Either change the current declaration to match the declaration of the tag, or create a new tag containing the different type.

IGNORETOKENS, # not in column 1 is ignored, skipping to end of line.

Description

In K & R mode, white space is not allowed before a preprocessing directive. The compiler will ignore this source line.

User Action

Either remove the white space or compile in a mode other than K & R.

IGNORSYSREG, Ignoring system register specified in routine's linkage.

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

IMAGINARYNA, The _Imaginary keyword is not supported by VSI C. It will be treated as an identifier in this compilation.

Description

Support for the _Imaginary keyword is an optional extension to the C standard. VSI C does not support this extension. All occurrences of _Imaginary will be treated as an identifier.

User Action

Do not use the _Imaginary type.

IMPFNCFALLOFF, The last statement in non-void function "*name*" is not a return statement.

Description

This message indicates that a function with an implicit return type of it does not end with a return statement. If function execution reaches the end of the function, the implied return statement that executes will return an undefined value. This might not have been what you intended.

User Action

Consider declaring the function to be a void function. If it is supposed to return a value, add a return statement with the value the function is to return.

IMPFNCMSSNGRET, Non-void function "*name*" with implicit return type int does not contain a return statement.

Description

This message indicates that a function with an implicit return type of int does not contain a return statement. This message is not issued for functions with an explicit return type. See message MISSINGRETURN.

User Action

Consider declaring the function to be a void function. If it is supposed to return a value, add a return statement with the value the function is to return.

IMPLICITFUNC, *context*the identifier "*name*" is implicitly declared as a function.

Description

A expression contained a reference to a function that has not been declared. The C99 standard requires that all referenced functions must be declared before they are referenced.

User Action

Declare the function before it is referenced.

INCARGTYP, Type of actual argument inconsistent with formal parameter declaration in *text*

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

INCARRAYPARM, *context*using array syntax to declare a parameter that is a pointer to an incomplete array type may not be portable.

Description

Although this array parameter declaration conforms to the C standard (since it is equivalent to a pointer to the inner array), other C compilers may not accept it.

User Action

Be aware of this difference if you plan to port this source to another compiler.

INCARRAYPARM1, *context*allowing an array parameter that has more than two unspecified element counts is a language extension.

Description

Because this parameter declaration will cause an array of incomplete types to be created, it does not conform to the C standard. Although some other C compilers will accept this declaration, many compilers will reject it.

User Action

Be aware of this difference if you plan to port this source to another compiler.

INCLUDENOPEA, A non-default pointer size or member alignment is specified, and the header files in *directory* are not protected. This might yield unpredictable results. The protect_headers_setup script can help. See the protect_headers_setup(8) man page for details.

Description

Using a non-default pointer size or member alignment can cause unpredictable results for system header files that are not protected and that rely on the default pointer size or alignment.

User Action

Examine the man page referenced in the message for more information.

INCLUDEOPEN, An error occurred while attempting to open the include file *name*: *problem*.

Description

An unexpected error occurred during the opening of an include file. The message text will contain additional information about the failure.

User Action

Correct the condition that caused the failure.

INCLUDEPROEPI, Cannot include files in a prologue or epilogue file.

Description

It is not possible for a prologue or epilogue file to perform an #include directive. This might lead to nested inclusion.

User Action

Remove the #include directive from the prologue/epilogue file.

INCOMPARRY, *context*the member *name* has incomplete array type. This is not strictly conformant with the C standard and might not be portable.

Description

The compiler has detected an array without a bounds specifier to be part of a struct or union type. The C89 standard does not allow members of this type. The C99 standard will allow only the final member of a struct with more than one named member to be of this type. Other C compilers might not successfully compile a program that uses this extension.

User Action

Specify the bounds if possible.

INCOMPARRY1, *context*the last member of a union, or a struct with only one named member, *name*, has incomplete array type. This is not strictly conformant with the C standard and might not be portable.

Description

The compiler has detected an array without a bounds specifier to be part of a struct or union type. The C89 standard does not allow members of this type. The C99 standard will allow only the final member of a struct with more than one named member to be of this type. Other C compilers might not successfully compile a program that uses this

User Action

Be aware of this extension if you wish to port the code.

INCOMPARRY2, *context*the last member of a struct with more than one named member, *name*, has incomplete array type. This does not conform to the C89 standard.

Description

The C89 standard does not allow struct members to be an array without a bounds specifier. The C99 standard will allow the final member of a struct with more than one named member to be an incompete type. Other C compilers may not support this C99 extension.

User Action

Be aware of this if you wish to port the code to a compiler that does not support C99.

INCOMPCALL, *context*the return type of "*expression*" is incomplete.

Description

A function with an incomplete return type other than void cannot be invoked.

User Action

Complete the function return type before the function is invoked.

INCOMPDEREF, *context*"*expression*" is a pointer to an incomplete struct or union and should not be used as the left operand of a member dereference.

Description

In certain modes, VSI C will allow the struct or union specifier of a member dereference operator (->) to specify a struct or union that does not contain the element specified by the right operand. While this is considered poor programming practice, it was common with older C compilers. In cases where the left operand is a pointer to an incomplete type, the practice is considered even worse. While VSI C will accept the construct in certain modes, the code should be modified. Further, this program does not conform to the C standard and might not be accepted by other C compilers.

User Action

Be aware of this if you wish to port the program.

INCOMPELINIT, *context*, an array's element type is incomplete, which precludes its initialization.

Description

In order to initialize an array, the array element type must not be incomplete.

User Action

Either remove the initializer or complete the array element type before this point in the program.

INCOMPELMNT, *context*the element type of an array type is incomplete.

Description

The element type of an array type is incomplete at the point in the program where the array is declared. While VSI C will allow this if the element type is completed later, other compilers might require the type to be complete at this point in the program.

User Action

Either complete the type before the array declaration, or be aware of this if you wish to port the program.

INCOMPMEM, The member "*name*" has an incomplete type.

Description

A struct or union member must not have an incomplete type. An exception is that VSI C will accept a member that is an array with unspecified bounds, although warnings are often generated for this case.

User Action

Complete the type before it is used in as a member of a struct or union.

INCOMPNOLINK, In this declaration, "*name*" has no linkage and is of an incomplete type.

Description

A declaration with no linkage cannot specify an incomplete type. Incomplete types can only be used for identifiers with external or internal linkage.

User Action

Either complete the type before the declaration or modify the declaration to specify an external or internal linkage.

INCOMPPARM, In the definition of the function "*function name*", the parameter "*parameter name*" has an incomplete type.

Description

This function definition contains a parameter with an incomplete type other than an array whose bounds are not specified. This is not valid.

User Action

Complete the type before the function definition.

INCOMPRETURN, In the definition of the function "*name*", the return type is an incomplete type other than void.

Description

A function definition cannot specify a return type that is an incomplete type except for the void type.

User Action

Complete the type before the function definition.

INCOMPSTAT, The static declaration of "*name*" is a tentative definition and specifies an incomplete type.

Description

This file scope static declaration declares an identifier with incomplete type. This is not valid because a static declaration will allocate storage for the object, but the object's size is not known at this point in the compilation.

User Action

Complete the type before the static declaration.

INCOMPSTATARR, Allowing the declaration of a static array with an incomplete type is a language extension.

Description

The VSI C compiler will allow an incomplete array type to appear in a static file scope declaration for compatibility with other compilers. This is an extension to the standard. Other compilers may reject this declaration.

User Action

Either use a complete type in this declaration, or change the storage class to extern.

INCOMPTENT, The type of the tentatively-defined variable "*name*" is incomplete at the end of the compilation unit.

Description

This file-scope declaration with no storage-class specifier declares an identifier with incomplete type. The type must be completed before the end of the compilation unit.

User Action

Complete the type.

INCOMPVALUE, *context*"*expression*" has incomplete type, and so cannot be used as an rvalue.

Description

It is not possible to get the value of an expression with incomplete type.

User Action

Complete the type before its value is used.

INCOMPVOID, *context*the element type of an array type is incomplete. The void type cannot be completed.

Description

The compiler has encountered an array with an element type of void. An array element must be an object type.

User Action

Change the type of the array element.

INCONSASSFUN, A function "*name*" appeared in more than one #pragma assert/hint func_attrs specifying the same assertion/hints.

Description

A function can appear on more than one #pragma assert or #pragma hint func_attrs as long as each #pragma specifies a different assertion/hint about the function. The assertion will be ignored.

User Action

Either remove the #pragma, or remove the function name from the pragma, or correct its spelling.

INITCONFLICT, Overlapping static storage initializations detected at Psect *text* + *number*

Description

The compiler back-end as detected a case where the same storage location has been initialized to more than one value. This can occur when inter-file optimization has been enabled.

User Action

Remove one of the initializers.

INITOVERLAP1, *context*, this initializer list will provide a value for a subobject that was initialized by the earlier initializer "*init*".

Description

This initializer list will provide a value for a subobject that has already been initialized. While this is valid, it might not have been what was intended.

User Action

Initialize each subobject only once.

INITVLA, A variable-length array declaration cannot contain an initializer. The initializer will be ignored.

Description

A variable-length array declaration cannot contain an initializer.

User Action

Initialize the array using assignment statements after the declaration.

INLINEIG, An inline specifier may only be used to declare an identifier for a function. The inline keyword will be ignored.

Description

The inline, __inline or __forceinline keywords have been used on a non-function type. Or a non-function type has been listed in a #pragma inline or #pragma forceinline directive

User Action

Remove the keyword or remove the identifier from the pragma.

INLINESTOCLSMOD, The *__inline or __forceinline* storage class modifier is a language extension and might not be portable.

Description

The __inline and __forceinline storage class modifiers are an extension of VSI C. Other C compilers might not successfully compile a program that uses the extension.

User Action

Be aware of this extension if you wish to port the code.

INPTRTYPE, *context*this argument to *function name* is of "*type name*" type and is not appropriate for the conversion specifier "*incorrect conversion*". The value may overwrite other data or produce unexpected results.

Description

The compiler has detected an input conversion specifier that does not match its corresponding argument. The corresponding argument may not be a pointer or may point to data that is wider or narrower than that specified by the conversion specifier. This might not have been what you intended.

User Action

Modify either the argument or the conversion specifier so that they match.

INSUFALN, Alignment specified for extern model is insufficient for variable. Extern model alignment updated.

Description

The current extern model places all external objects in a section whose alignment is not sufficient for the alignment of an object being placed in that section. The compiler will update the alignment of the section so that it is adequate for the object.

User Action

Either increase the alignment of the section or move the object to another section.

INTBADLINKAGE, #pragma use_linkage was applied to the intrinsic function "*routine name*". The function will be treated as an ordinary external function.

Description

Trying to optimize a pointer argument passed to an intrinsic function, the compiler discovered that #pragma use_linkage had been applied to the function declaration. The intrinsic function of this name that is understood by the compiler does not allow you specify a linkage. Therefore the compiler must assume that you are supplying your own function definition, and treat this as a call to an external function with no special properties.

User Action

If you want to call the intrinsic function, remove the #pragma use_linkage directive. If you are supplying your own function definition, you may want to rename the function or add a #pragma function directive for it.

INTCONCASTSGN, *context*casting of the constant "*constant*" to *type* type will cause a change in sign.

Description

Either a negative constant value has been cast to an unsigned type, or a positive value has been cast to a signed type and will be treated as a negative number after the cast.

User Action

Change the constant so that the sign will match the type of the cast.

INTCONCASTTRU, *context*casting of the constant "*constant*" to *type* type will cause data loss.

Description

A constant is cast to a type that is too small to hold the constant value. Data will be lost in the conversion.

User Action

Remove the cast, or use a smaller constant.

INTCONST, Ill-formed integer constant.

Description

An invalid integer constant was encountered.

User Action

Correct the integer constant.

INTCONSTSIGN, *context*conversion of the constant "*constant*" to *type* type will cause a change in sign.

Description

Either an unsigned type was assigned a negative constant value, or a signed type was assigned a positive contant value which will be evalated as a negative number after the assignment. Note that this message is not output for assignments to 1-bit bitfields. The message bitconstsign is generated in that case.

User Action

If this is what you intended, cast the constant to the desired type. You might also want to change the constant to the correct signed or unsigned value in order to avoid the optional message intconcastsgn, which reports sign changes caused by casts.

INTCONSTSIGNED, This integer constant value will be given the type long long int. This is compatible with the C99 standard. Older versions of the compiler would have given this unsigned long int type.

Description

With the introduction of the long long int type, the C99 standard changed the rules for how the type of certain integer constants are determined. Unsuffixed decimal constants which are too large for long int, but could fit in an unsigned long int are given the type long long int in C99. Prior to C99 these would be given unsigned long int type.

User Action

Be aware of this difference.

INTCONSTTOOBIG, This integer constant is too large for the long long type. It will be given the unsigned long long type.

Description

The C99 standard specifies that a decimal constant must fit in a signed type. This constant is too large for the long long int type. For compatibility with older versions of the compiler, the constant will be given the unsigned long long type.

User Action

Append a 'U' suffix to the constant. This will force it to be unsigned.

INTCONSTTRUNC, *context*conversion of the constant "*constant*" to *type* type will cause data loss.

Description

A constant is converted to a type that is too small to hold the constant value. Data will be lost in the conversion.

User Action

If this is what you intended, cast the constant to the desired type. You might also want to mask off the high-order bits before casting in order to avoid optional message intconcasttru, which reports data loss caused by casts.

INTCONSTUNSIGN, This integer constant value will be given the type unsigned long int. This is compatible with the C89 standard and older compilers. The C99 standard requires this to be a signed long long int.

Description

With the introduction of the long long int type, the C99 standard changed the rules for how the type of certain integer constants are determined. Unsuffixed decimal constants which are too large for long int, but could fit in an unsigned long int are given the type long long int in C99. Prior to C99 these would be given unsigned long int type.

User Action

Be aware of this difference.

INTERNALPRAGMA, This is an internal pragma which should only be used by the compiler development team. It should not appear in user programs as it may cause unexpected behavior.

Description

This pragma exists only to allow the compiler developers to test certain functionality of the compiler. Its use outside the development team is unsupported.

User Action

Remove the pragma.

INTIMPLIED, In the declaration of "*name*", no type was specified. Type defaulted to int. This is a violation of the C99 standard.

Description

The declaration contains a storage-class specifier, but no type was specified. The compiler will assume a type of int. Omitting the type specifier is not valid in C++ or in C99, and is often considered poor programming practice.

User Action

Add a type specifier to the declaration.

INTOVERFL, *context*integer overflow occurs in evaluating the expression "*expression*".

Description

An integer overflow occurred while evaluating a constant expression. The value of the expression is undefined.

User Action

Correct the constant expression so that it does not overflow.

INTRINSICCALL, *context*an apparent invocation of intrinsic function "*name*", *problem*. It will be treated as an ordinary external call.

Description

A function that could be handled internally by the compiler has been called in a manner that is inconsistent with expected usage. In such a case, the compiler will generate a run-time call to the function. This could result in performance loss.

User Action

If the function is intended to refer to the runtime library routine, the appropriate header file should be included in the source to provide the full function prototype and allow certain types of argument conversions. Alternatively, call arguments could be type cast as specified in the error message, or the function prototype could be added by hand. If the function is not intended to refer to the runtime library routine, the intrinsic version can be disabled by means of the "#pragma function (function_name)" directive.

INTRINSICDECL, *context*the declaration for intrinsic function "*name*" referenced at *location*, *problem*. It will be treated as an ordinary external function.

Description

A function that could be handled internally by the compiler has been declared with a prototype that does not agree with what the compiler expected to see, or has been declared at block scope instead of file scope. The function might in fact be a similarly-named replacement for the expected function, or the prototype might be incorrect or misplaced. In such cases, the function will not be handled internally, but will instead be called at run time in the usual manner. This could result in a performance loss.

User Action

If the function is intended to refer to the runtime library routine, the appropriate header file should be included in the source (note that it is not portable to include standard headers at other than file scope). Alternatively, the prototype could be modified as specified in the error message. If the function is intended to be a replacement for the runtime library routine, disable the intrinsic version by specifying "#pragma function(function_name)" in the source file.

INTRINSICDECLER, *context*the declaration for the prototyped intrinsic function "*name*" is incorrect: *problem*.

Description

A function that could be handled internally by the compiler and requires a prototype, has been declared with a prototype that does not agree with what the compiler expected to see. The function might be intended as a similarly-named replacement for the compiler-known function, or the prototype might be incorrect. The source must be modified to specify the intended behavior.

User Action

If the function is intended to refer to the compiler-known routine, the appropriate header file should be included in the source. Alternatively, the prototype could be modified as specified in the error message. If the function is intended to be a replacement for the compiler-known routine, disable the intrinsic version by specifying "#pragma function(function_name)" in the source file.

INTRINSICINT, *context*the *place* type for intrinsic "*name*" is being changed from "size_t" to "int".

Description

A function that is handled internally by the compiler expects an argument type or return type of "size_t", but the prototype for the function uses "int". The compiler will use "int" in this case.

User Action

Declare the function by including the appropriate header file. Alternatively, provide a private declaration (or modify an existing private declaration) with "size_t" in the appropriate location(s), and with "size_t" defined as it is in the standard system header files. If the function is not intended to refer to the runtime library routine, the intrinsic version can be disabled by means of the "#pragma function (function_name)" directive.

INTUNDFUN, There is no function declaration visible for the identifier "*name*" at the point of this #pragma *pragma type*.

Description

An identifier specified in a #pragma intrinsic or #pragma function directive must refer to a function declaration visible at the point of the pragma. The identifier will be ignored.

User Action

Either remove the identifer from the pragma, correct its spelling, or reorder the source to ensure that a declaration of the identifier as a function is visible at the point of the pragma.

INVALIDARG, Invalid argument to *pragma* pragma. Pragma is ignored.

Description

An invalid argument has been specified for a pragma directive. The compiler will ignore the directive.

User Action

Correct the directive.

INVALIDSTR, The # operator produced an invalid string.

Description

During the expansion of a macro, the # stringize operator produced a token that is not a valid string. The operand to the stringize operator must contain characters that form a valid string.

User Action

Correct the operand to the stringize operator.

INVALTOKEN, Invalid token discarded.

Description

An unexpected token was encountered by the compiler. The token has been ignored. An example is the preprocessing operator "#" appearing outside a macro body (int #a;).

User Action

Remove the unexpected token.

INVCPPINARGS, Possible directive "*#directive*" within a macro argument list. The directive is treated as part of the argument list, and not as a preprocessing directive.

Description

The compiler has encountered a directive as part of the argument list of a macro invocation. This directive will be treated as part of the argument list, and not as a preprocessing directive. The behavior might be different than other compilers.

User Action

Rewrite the macro invocation so that it does not include the directive.

INVDUPENUM, *context*the value of the enumerator "*name*" conflicts with a previous declaration.

Description

The specified enumerator name has been previously declared with a different value.

User Action

Either use a different enumerator name or remove the previous declaration of the name.

INVNOMEMPRAG, Invalid argument to nomember_alignment pragma. Pragma is ignored.

Description

The compiler was unable to parse a #pragma nomember_alignment directive. The directive will be ignored.

User Action

Correct the directive.

INVPACKPRAG, Invalid pack pragma. Pragma is ignored.

Description

The compiler was unable to parse a #pragma pack directive. The directive will be ignored.

User Action

Correct the directive.

INVPPDIRPEA, The preprocessor directive *name* is not allowed in a prologue or epilogue file. The directive is ignored.

Description

It is not possible for a prologue or epilogue file to have this preprocessor directive in it.

User Action

Remove the offending preprocessor directive from the prologue/epilogue file.

INVSTATIC1, *context* the keyword "static" and/or type qualifiers may appear only in the outermost array-bounds specifier of a function parameter. Keyword/qualifier ignored.

Description

The keyword "static" or a type specifier appeared in an array-bound specifier that was either not part of the declarator for a function parameter or it was not the outermost array-bound specifier of a function parameter.

User Action

Remove the keywords or confine them to use in the outermost array-bound specifier of a function parameter.

INVSTATIC3, The keyword "static" may not appear in an array-bound specifier for a declaration of an array of unknown size. Keyword ignored.

Description

The keyword "static" appeared in the declaration of an array whose size was not known, either because array has in incomplete type or because the array has a star bounds specifier.

User Action

Remove the keyword.

INVSTATIC4, An expression specifying the bound is required when the keyword "static" is used in an array-bounds specifer. Keyword ignored.

Description

The keyword "static" appeared in an array-bounds specifier that did not have an expression describing the array bound. The keyword tells the compiler that actual arguments passed to this parameter will always have at least as many elements as specified in the formal parameter. It is inconsistent to specify the keyword without also supplying a value for the bound.

User Action

Remove the keyword or supply a value for the bound.

INVSTATIC5, *context* the static bound value differs from the static bound value in another declaration at *location*. The smaller static bound value will be used.

Description

In a previous declaration of a function one or more parameters with array type were declared with a different static bound value than in the current declaration. This can occur if one of the sizes of the corresponding static arrays differ between the two declarations.

User Action

Remove the keyword "static" from the declarations, or give all function declarations the same static bound value.

INVSTATIC6, *context* neither the keyword "static" nor a type qualifier may be used in array-bounds for old-style function parameters. Keyword/qualifiers ignored.

Description

Use of the keyword "static" or a type specifier within the outermost array bound specifier of a formal parameter is a new feature in the C99 statndard. It cannot be used in old-style function definitions.

User Action

Remove the keywords or convert the code to use prototype-style function declarations and definitions.

IVDEPNOFOR, This #pragma directive was not followed by a for statement. The directive will be ignored.

Description

The #pragma ivdep and #pragma unroll directives modify the for loop which follows them. The compile has encoutered one of the directives without a following for loop. The directive will be ignored.

User Action

Remove the directive.

KEYCOMB, Illegal combination of keywords.

Description

An invalid combination of Microsoft keywords was encountered during a declaration. In most cases this is because the keywords contradict each other. One example would be using the __fastcall and __stdcall modifiers in the same function declaration. This message is only output when the compiler is in Microsoft mode.

User Action

Remove one of the contradictory modifiers.

KNRFUNC, The function "*name*" is defined using the old style K&R syntax. The C standard has marked this syntax as obsolescent, and it is not supported in C++. Consider using the standard C prototype syntax.

Description

The function uses an old style function definition. VSI recommends that old style function definitions be replaced by prototype-format definitions.

User Action

Recode the function definition to use the recommended prototype-format definition.

LABELWOSTMT, Accepting a label without a following statement is a language extension.

Description

The C standard states that a label must be followed by a statement. For compatibility with other C compilers, VSI C will accept a label without a statement. Be aware that this program does not conform to the standard and may be rejected by other compilers.

User Action

Add a semicolon after the label to create a null statement.

LCRXCOND, Common Data Dictionary description extraction condition. *msg*.

Description

Something went wrong while trying to get the CDD record description from the CDD. The error message that follows gives more information about the nature of the problem.

User Action

If necessary, correct the indicated condition in the CDD record description or with the user environment.

LDCOMPLEXNYI, *context*the type long double _Complex is not fully supported on this platform. The type is only accepted when the compilation specifies the option to make the long double type 64-bits in size.

Description

On some platforms VSI C does not support the long double _Complex type where the real and imaginary component are 128-bits in size. As VSI C requires that each component of a long double _Complex be the same size as a long double, this compilation must specify the option to treat long double as 64-bits.

User Action

Either specify the correct compiler option or use the double _Complex type instead of the long double _Complex type.

LEXNESTPAR, Lexically nested parallel at scope *text* is not supported

Description

Nested parallel directives are not supported.

User Action

Remove the nested parallel directive.

LISTOPEN, An error occurred while attempting to open the listing file: *reason*.

Description

An unexpected error occurred during the creation of the listing file. The message text will contain additional information about the failure.

User Action

Correct the condition that caused the failure.

LOCALEXTINI, The block-level declaration of "*name*" includes an initializer and specifies storage class extern.

Description

A block-level declaration with extern storage class cannot contain an initializer.

User Action

Remove the initializer from the declaration or move the declaration to file scope.

LONGDEBUG, The identifier name exceeds *number* characters; name passed to the debugger will be truncated to "*truncated spelling*".

Description

On some platforms, the name length supported by the compiler is greater than the length supported by the debugger. In this case the compiler must truncate the name when it is output to the debugger symbol table for this compilation.

User Action

Reduce the size of the name.

LONGDOUBLENY1, *context*type long double has the same representation as type double on this platform and is treated as a synonym for type double in this compilation mode.

Description

VSI C does not support the long double type on this platform. In this compilation mode, the compiler will treat the long double type as a synonym for the double type.

User Action

Be aware of this.

LONGDOUBLENYI, *context*type long double has the same representation as type double on this platform.

Description

Although VSI C will recognize the long double type as a different type than double in this compilation mode, on this platform they will both use the same representation. Using long double will not provide any additional precision or range.

User Action

Be aware of this.

LONGEXTERN, The external identifier name exceeds *number* characters; truncated to "*truncated spelling*".

Description

The length of an identifier with external linkage exceeds the maximum allowed on this platform. The name used in an output object file will be truncated to meet the platform restrictions. Note that the debugger name will be unchanged.

User Action

Reduce the size of the name. On OpenVMS platforms the /NAMES=SHORTENED qualifier can also be used. When the qualifier is specified, the compiler will encode long external names instead of truncating them.

LONGFLOATEXT, *context*long float as a synonym for double is a language extension.

Description

Certain standard modes allow the use of the long float type as a synonym for double. This is not allowed by the C standard. This message indicates this use of long float as a potential portability problem.

User Action

Change long float to double.

LONGLINE, A *type* source line longer than *number* characters was encountered.

Description

The length of a source line has exceeded the maximum length supported by the VSI C compiler.

User Action

Reduce the size of the line.

LONGLONGSUFX, The integer constant is of type "*type*", which is a new feature of C99 might not be portable.

Description

The use of the suffix ULL or LL on an integer constant does not conform to the C89 standard and might not be accepted by other C compilers.

User Action

Be aware of this if you wish to port the program.

LONGLONGTYPE, *context*type "*type*" is a new feature in C99.

Description

On some platforms, VSI C will accept the [unsigned] long long type as a way to declare [unsigned] 64-bit integers. The long long int type is a new feature of C99 and other compilers might not accept this declaration.

User Action

Be aware of this portability concern.

LONGMODULEID, Identifier "*name*" in a #pragma module or #module directive exceeds 31 characters.

Description

A module or identification name specified in the #pragma module or #module directive must be less than 32 characters. The compiler will truncate the name to the first 31 characters specified.

User Action

Shorten the module or identification name.

LONGMODULESTR, The identification string *string* in a #pragma module or #module directive exceeds 31 characters. The compiler will ignore the directive.

Description

An identification string specified in the #pragma module or #module directive must be less than 32 characters. The compiler will ignore the directive.

User Action

Shorten the identification string.

LONGPREFIX, Prefix string too long. Truncated to "*newprefix*".

Description

The specified prefix to the #pragma extern_prefix directive is too large for this platform. The prefix will be truncated.

User Action

Reduce the size of the specified extern prefix.

LONGPSECT, Psect name is too long (maximum is 31 characters).

Description

The psect name specified in a globaldef declaration was longer than 31 characters. This exceeds the maximum allowed length.

User Action

Either reduce the psect name to 31 characters or remove the psect specifier.

LONGTOKEN, An individual token longer than *number* characters was encountered.

Description

The length of an individual token has exceeded the maximum length supported by the VSI C compiler.

User Action

Reduce the size of the token; perhaps it can be converted into two or more smaller tokens.

LVALUECAST, *context*the result of the cast "*cast*" is used as an lvalue.

Description

The result of a cast has been used as an lvalue. This is a language extension of VSI C. The program does not conform to the C standard, and might not be accepted by other compilers.

User Action

Remove the cast.

MACROREDEF, The redefinition of the macro "*name*" conflicts with a current definition because *reason*. The redefinition is now in effect.

Description

A macro has been redefined with either different formal parameters and/or a different body than a previous definition of the macro.

User Action

Either make all definitions of the same macro identical, or undefine the macro using the #undef preprocessing directive before it is redefined.

MACROREDEFIN, Macro redefined.

Description

A #define preprocessing directive has redefined a macro whose previous definition contained an error or warning. Normally, the compiler will issue a warning if a macro is redefined to something other than the previous definition. However, if the previous definition caused a warning or error to be generated, this informational message is output instead.

User Action

Do not redefine a macro without first undefining it.

MAINNOTINT, Strict standard C extension: The declaration of the "main" function has a return type other than int.

Description

Standard C requires that the "main" function be defined with a return type of int. VSI C will accept other return types, but the program does not conform to the C standard. The status value returned to the environment may not be what you expect, and other C compilers may not accept the definition as written.

User Action

Define the "main" function with a return type of int for maximal portability.

MAINPARM, Strict standard C extension: The declaration of the "main" function has more than two parameters.

Description

Standard C requires that the "main" function takes no more than two parameters. VSI C will accept more, but the program does not conform to the C standard.

User Action

Modify the declaration if you want the program to be standard conformant.

MAINPROGEXT, MAIN_PROGRAM is a language extension.

Description

The use of MAIN_PROGRAM to designate a function as the main program is a language extension of VSI C. Other C compilers might not successfully compile a program that uses the extension.

User Action

The main program should be declared by naming the function main.

MAPREGIGNORED, The linkage register "*registers*" has no effect on Alpha and will not be mapped to any register on IA-64. This condition may cause the SHOWMAPLINKAGE message output for this directive to be incorrect.

Description

The use of an Alpha argument register (R16-R21) in a linkage characteristic other than "parameters" or "results" has no effect on Alpha. No mapping to an IA64 register will be done for this register. This may cause the mapped linkage shown in the showmaplinkage message to be incorrect.

User Action

Remove the register from the characteristic.

MATHERRNO, *context*function *name* is defined to set errno when a domain error or range error occurs. As an intrinsic, it may not be able to do so.

Description

Any code that tests the value of errno set by this function may not work properly due to the optimizations that are possible when this function is an intrinsic.

User Action

If the value of errno set by this function is ignored, tell the compiler via its command line qualifiers to assume nomath_errno. Otherwise, disable the intrinsic by using a #pragma function(func-name).

MAYHIDELOSS, *context*"*expression*" has a larger data size than "*target type*". The use of a cast operator can suppress the message that this assignment might result in data loss.

Description

In a cast of a pointer to one of the integer types, or a cast of one of the integer types to a pointer, or a cast of one pointer type to another, the size of the source is greater than the size of the type being cast to. This cast could result in a loss of data if it is used as the source of an assignment. This potential loss of data can be verified by removing the cast and seeing if the compiler emits a loss of data message on the assignment.

User Action

If the cast cannot lose precision, it is safe to ignore this warning.

MAYLOSEDATA, *context*"*expression*" has a larger data size than "*target type*". Assignment can result in data loss.

Description

In an assignment of a pointer to one of the integer types, or one of the integer types to a pointer, the size of the source is greater than the size of the destination. The assignment can result in a loss of data. This might not have been what you intended.

User Action

If this was the intended operation, cast the source to the type of the destination before the assignment.

MAYLOSEDATA2, *context*"*expression*" has a larger data size than "*target type*". Assignment can result in data loss.

Description

In an assignment of two pointers, the size of the source is greater than the size of the destination. The assignment can result in a loss of data. This might not have been what you intended.

User Action

If this was the intended operation, cast the source to the type of the destination before the assignment.

MECHMISMATCH, Argument passing mechanism does not match formal parameter mechanism for *text*

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

MEMBERVLA, *context*the struct or union member "*name*" cannot be declared with a variably modified type.

Description

Only ordinary identifiers with block scope and without storage class extern, or ordinary identifiers with function prototype scope can be declared with a variably modified type.

User Action

Correct the declaration.

MISALGNDMEM, This member is at offset *offset*, which is not a multiple of the member's alignment of *align*. Consider padding before this member, rearranging the order of member declarations, or using #pragma member_alignment.

Description

A member of a struct or union requires an alignment for efficient access but will be allocated at an offset that is not a multiple of that alignment.

User Action

Use one of the suggestions made in the message.

MISALGNDSTRCT, This member requires *align1* alignment for efficient access, but is contained in a struct containing *align2* alignment. Consider using #pragma nomember_alignment *align1*.

Description

A member of a struct or union requires an alignment for efficient access that is more strict than the alignment of the enclosing struct or union. Even though this member is correctly aligned within the struct or union, if the struct or union is enclosed within another type, the member in question might be placed at a position with incorrect alignment for its type.

User Action

Use either the #pragma member_alignment directive or #pragma nomember_alignment directive with an argument equal to or greater than the alignment of the member.

MISDEFARG, Missing argument for "defined" operator.

Description

The defined preprocessing operator was not supplied with an argument. The operator expects an identifier optionally enclosed in parenthesis. The value of the operator is undefined.

User Action

Supply a valid argument to the preprocessing operator.

MISDEFPAR, Missing right parenthesis for "defined" operator.

Description

The defined preprocessing operator began with a left parenthesis, but no matching right parenthesis was found. The value of the operator is undefined.

User Action

Add the right parenthesis after the preprocessing operator argument.

MISMATPARAM, *context*parameter *number* has a different type than specified in an earlier declaration of this function.

Description

A function has been redeclared with a parameter whose type is different than the type specified in a previous declaration of the function. This generally leads to additional errors.

User Action

Correct the function declarations so that the parameter information is the same in each declaration.

MISMATPRSRET, "*name*" has a non-void return type but its linkage "*name*" preserves the return register(s). Standard linkage will be used.

Description

The function or typedef's special linkage specifies that the register(s) used to return the function value are to be preserved. This is invalid as the register can either be preserved, or hold the return value, but not both.

User Action

Modify the #pragma linkage directive to either remove the register from the preserved list or to specify another register to hold the return value.

MISMATTHREAD, *context*the __declspec(thread) storage class modifier of "*name*" is different from a previous declaration of "*name*" at *location*.

Description

If an object is declared with thread-local storage, then all declarations of that object must declare it with thread-local storage.

User Action

Either remove the invalid redeclaration or modify it to match the previous declaration.

MISPARAMCOUNT, *context*the number of parameters differs from an earlier declaration of this function.

Description

A function has been redeclared with a different number of parameters than a previous declaration of the function. This message generally proceeds additional errors.

User Action

Correct the function declarations so that the parameter information is the same in each declaration.

MISSINGCASE, Was the 'case' keyword omitted? Within a switch statement, "*label*" defines an unreferenced label that matches an enumeration constant.

Description

This user label has been defined, but there are no references to it. As the label is defined inside a switch statement, and the label name matches an enumumeration constant name, there is a chance you intended this to be a case label.

User Action

Remove the label or add the 'case' keyword before it.

MISSINGCOMMA, This parameter is not preceded by a comma.

Description

The compiler has encountered a parameter specifier that is missing a preceding comma. The parameter will be defined anyway, though this may not have been what you intended.

User Action

Correct the formal parameter list so that it consists of a comma separated list of identifiers (possibly followed by ", ...").

MISSINGFUNC, The function "*name*" has internal linkage, occurs in a context that requires its definition, and has no definition.

Description

The program has referenced a function declared with static storage class, but the function is not defined in the compilation unit. If a program references a static function, the function must be defined in the compilation unit.

User Action

Either define the function or change the function declaration to have extern storage class.

MISSINGLABEL, The label "*label name*" is the target of a goto statement within the function "*function name*", but has no definition within "*function name*".

Description

Every label referenced in a goto statement must be defined in the same function.

User Action

Either change the name of the label in the goto statement, or define the label.

MISSINGRETURN, Non-void function "*name*" does not contain a return statement.

Description

This message indicates that a function with an explicit return type does not contain a return statement. This message is not issued for functions with an implicit return type of int. See message IMPFNCMSSNGRET.

User Action

Consider declaring the function to be a void function. If it is supposed to return a value, add a return statement with the value the function is to return.

MISSINGTYPE, Missing type specifier or type qualifier.

Description

The compiler was expecting a type specifier or type qualifier, but one was not found.

User Action

Correct the program syntax.

MISSPELLDEF, The user label "*label*", defined within a switch statement, is never referenced.

Description

This user label has been defined, but there are no references to it. As the label is defined inside a switch statement, there is a chance this is a misspelling of "default".

User Action

Remove the label or correct the spelling.

MIXALLOCAVLA, *context*this call to __ALLOCA occurs in a block that contains *vlaallocafrag1*. The storage allocated by this __ALLOCA call will *vlaallocafrag2*vla or aligned automatic declaration was at *where*.

Description

Storage allocated for arrays of variable length and for automatics whose alignment is greater than octaword have their storage deallocated when the block they are declared in exits. Storage allocated by __ALLOCA is not normally deallocated until function exit. VSI C cannot support both types of deallocation in the same block. Therefore, when both appear in the same block, the storage for both will be deallocated with the block exits.

User Action

Be aware of this. If the storage allocated for __ALLOCA must remain allocated until function exit, move the __ALLOCA call outside the block declaring the vla or the aligned auto.

MIXFUNCVOID, *context*compatibility of a pointer to void and a pointer to a function is not portable under the C standard.

Description

The C standard defines pointer to void as being assignment compatible only with pointers to object or incomplete types. An implementation may represent function pointers in a way that cannot be stored in a pointer to void (or vice-versa). Thus even an explicit cast between a function pointer and a pointer to void is not portable.

User Action

If a generic pointer to function is needed, declare a typedef for some pointer to function type, and always use explicit casts to assign to and from that type.

MIXINLINE, The function *name* is declared both *this* and *that*.

Description

A function is declared with more than one of the forceinline, inline, or noinline attributes. It will be given the attribute that will provide the most optimization.

User Action

Make sure each function has only one of the attributes.

MIXLINKAGE, *context*"*name*" is declared with both internal and external linkage. The previous declaration is at *location*.

Description

This warning is output in certain cases when the linkage of a declaration conflicts with the linkage specified in an earlier declaration.

User Action

Change one of the declarations so that the linkages match.

MIXLINKAGE1, *context*"*name*" is declared with both internal and external linkage. The previous declaration is at *location*.

Description

This informational is output when a function previously declared to have extern storage class is redeclared to have internal storage class and the mode of the compiler is common (K & R) mode.

User Action

Change one of the declarations so that the linkages match.

MIXOLDNEW, The definition of the function *name* includes both a prototype and a declaration list.

Description

A function has been defined using both a declaration list and a prototype. This is not valid.

User Action

Correct the declaration.

MIXSTORCLS, *context*a storage class has already been specified. This storage class is ignored.

Description

The same declaration contains more than one storage class specifier. The compiler will ignore all storage class specifiers after the first one.

User Action

Change the declaration to use only one storage class specifier.

MIXVLAALLOCA, Declaring *vlaallocafrag1* in the same block as a call to __ALLOCA will cause the storage allocated by any __ALLOCA call to *vlaallocafrag2*previous call to __ALLOCA was at *where*.

Description

Storage allocated for arrays of variable length and for automatics whose alignment is greater than octaword have their storage deallocated when the block they are declared in exits. Storage allocated by __ALLOCA is not normally deallocated until function exit. VSI C can not support both types of deallocation in the same block. Therefore, when both appear in the same block, the storage for both will be deallocated with the block exits.

User Action

Be aware of this. If the storage allocated for __ALLOCA must remain allocated until function exit, move the __ALLOCA call outside the block declaring the vla or the aligned auto.

MODNOIDSTR, Invalid identifier or character-string constant specification.

Description

If specified, the second argument to the #pragma module or #module directive must be either an identifier or a string constant.

User Action

Correct the directive.

MODSTORCLS, Storage class modifier noshare has no meaning with this storage class. Modifier is ignored.

Description

The storage class modifier noshare is only valid for variables with a storage class of static, extern, or globaldef. It is ignored for other storage classes.

User Action

Remove the noshare storage class modifier.

MODULEFIRST, "#pragma module" or "#module" directive must precede any language text.

Description

The #pragma module or #module directive must appear before any declarations. The directive will be ignored.

User Action

Move the directive to the top of the compilation unit.

MSGPOP, This "restore" has underflowed the message stack. No corresponding "save" was found.

Description

The message stack, managed by the #pragma message and #pragma environment directives, contains more restores than saves. This could signify a coding or logic error in the program.

User Action

Make sure each restore has a corresponding save.

MSGSFRMEXLCODE, Enabling this message may cause additional messages from excluded code to be output.

Description

This message is never output by the compiler. Instead it is used to control whether other messages will be output. Normally, the compiler will not output some messages when it is processing code that it knows will never be executed. One example of this would be the second operand of the conditional operator when the first operand is FALSE. This suppression of these messages can be overridden by enabling this message.

User Action

Decide if you want the additional messages.

MULTICHAR, A character constant includes more than one character or wide character.

Description

A character constant includes more than one character. While this is valid, it might not have been what you intended.

User Action

Verify that the constant should contain more than one character.

MULTILINK, Multiple linkage pragmas specified for "*routine name*".

Description

The same routine appeared in more than one #pragma use_linkage directive. Each routine can only be given one linkage.

User Action

Remove the routine from all but one #pragma use_linkage directive.

MULTILINKREG, The register "*register*" is specified more than once in the linkage pragma. Pragma is ignored.

Description

The same register was specified more than once in the same register list in a #pragma linkage directive. The compiler will ignore the entire pragma.

User Action

Correct the directive.

MULTIMAIN, More than one main program has been defined.

Description

The compiler has encountered more than one main program in this compilation unit. Each program can have only one main program.

User Action

Remove one of the main programs.

MULTIPSECTNAME, Multiple *psect_type* names specified. The name "*new_name*" supersedes "*old_name*".

Description

More than one #pragma code_psect or #pragma linkage_psect was encountered. The psect specified by the later #pragma supersedes the one specified earlier. This message is only output for C compilers on OpenVMS Alpha.

User Action

Each program should contain at most one #pragma code_psect and one #pragma linkage_psect.

NAMESHORTENED, The external identifier or module name "*name*" exceeds 31 characters. The name has been shortened to "*shortened spelling*".

Description

A compilation that used the /NAMES=SHORTENED qualifier or #pragma names shortened directive has encountered a name that needs to be shortened. The external name will be different than the internal name. Also, because the external name exceeds the length specified by standard C as the minimum external length an implementation must support, this program does not strictly conform to standard C and might not be accepted by other C compilers.

User Action

Be aware of these items.

NAMESLOWER1, The /NAMES=LOWERCASE qualifier is no longer supported. The qualifier /NAMES=AS_IS will be used.

Description

While the C language has always required identifiers with internal linkage to be treated case sensitively. It traditionally permitted implementations to monocase identifiers with external linkage. Modern standards require C/C++ implementations to preserve the case of identifiers with external linkage. As VMS and other operating systems that traditionally implemented monocasing chose uppercase as the convention, /NAMES=LOWERCASE runs contrary both to the C and C++ standards and to traditional conventions. Continued support for this option interferes with support for compatibility between old code compiled with /NAMES=UPPERCASE and new code compiled with /NAMES=AS_IS.

User Action

Use /NAMES=AS_IS, making source code changes as needed.

NEEDADDRCONT, *context*"*name*" does not have a constant address, but occurs in a context that requires an address constant.

Description

A variable with static storage has been initialized to the address of an object whose address is not constant. This can happen if a static pointer variable is initialized to the address of an automatic variable.

User Action

Either make the initialize a constant, or, if possible, initialize the static storage using a run-time assignment.

NEEDARITH, *context*"*expression*" has *type* type, which is not arithmetic.

Description

An expression that must be an arithmetic type was not an arithmetic type. For example, the operands of an arithmetic operator such as * must be arithmetic type.

User Action

Modify the expression so that it is an arithmetic type.

NEEDCONSTEXPR, *context*"*name*" is not constant, but occurs in a context that requires a constant expression.

Description

An expression that must evaluate to a compile-time is not a constant.

User Action

Modify the constant expression so that it will evaluate as a compile-time constant.

NEEDCONSTEXT, *context*"*name*" is not constant, but occurs in a context that requires a constant expression. This is an extension of the language.

Description

The C89 standard requires that an initializer for an automatic aggregate or union type object have an initializer that is a list of constant expressions. VSI C allows non-constants in these initializers. This is an extension to C89. Although this is allowed by the C99 standard, other C compilers might not successfully compile a program that uses this extension.

User Action

Be aware of this if you wish to port the program.

NEEDDFLOAT, The CDD description for *name* specifies the D_Floating data type. The data can only be represented when compiling with /FLOAT=D_FLOAT.

Description

The /FLOAT command-line qualifier specified a floating type other than D_floating format. The CDD description specified was D_floating type, which did not match the floating type specified on the command line.

User Action

Specify the correct command-line qualifier, or change the description of the item in the CDD.

NEEDFUNCPTR, *context*"*expression*" points to *type* type, but occurs in a context that requires a pointer to a function type.

Description

An expression that must be a pointer to a function type is a pointer to an object or incomplete type. For example, if a function invocation expression is a pointer, it must be a pointer to a function type.

User Action

Modify the expression so that it is a pointer to a function type.

NEEDGFLOAT, The CDD description for *name* specifies the G_Floating data type. The data can only be represented when compiling with /FLOAT=G_FLOAT.

Description

The /FLOAT command-line qualifier specified a floating type other than G_floating format. The CDD description specified was G_floating type, which did not match the floating type specified on the command line.

User Action

Specify the correct command line qualifier, or change the description of the item in the CDD.

NEEDIEEE, The CDD description for *name* specifies a VAX floating data type. The data cannot be represented when compiling with /FLOAT=IEEE_FLOAT.

Description

The command-line qualifier /FLOAT=IEEE_FLOAT was specified, indicating that all floating-point data should be represented in IEEE-floating format, yet the CDD description specified a non-IEEE_floating type.

User Action

Specify the correct command-line qualifier, or change the description of the item in the CDD.

NEEDIEEE1, The CDD description for *name* specifies an IEEE floating data type. The data can only be represented when compiling with /FLOAT=IEEE_FLOAT.

Description

The CDD description for an item specifies an IEEE floating point type. However this module was not compiled with the /FLOAT=IEEE_FLOAT qualifier.

User Action

Specify the correct command-line qualifier, or change the description of the item in the CDD.

NEEDINTEXPR, *context*"*expression*" has *type* type, which is not integral.

Description

An expression that must be an integer type was not integral. For example, an array-index specifier must be an integral type.

User Action

Modify the expression so that it is an integral type.

NEEDLVALUE, *context*"*expression*" is not an lvalue, but occurs in a context that requires one.

Description

An expression that must be an lvalue was not an lvalue. For example, the operand of the address-of operator must be an lvalue.

User Action

Modify the expression so that it is an lvalue.

NEEDMEMBER, *context*"*name*" is not a member of "*struct or union expression*".

Description

The second operand of a . or -> operator specifies a member name that is not a member of the struct or union type specified by the first operand. Note that in certain modes, VSI C will search all other visible struct/union types for a matching member name. If it finds one, a diagnostic will be issued, and the offset of that name will be used.

User Action

Specify a valid member name.

NEEDNONBLTIN, *context*"*name*" is a builtin and cannot be used in this context.

Description

A program has used a builtin function in a way that is invalid for builtin functions. For example, a program cannot take the address of a builtin.

User Action

Remove the improper use of the builtin.

NEEDNONCONST, *context*"*expression*" has const-qualified type, but occurs in a context that requires a modifiable lvalue.

Description

The code has attempted to modify an object that is either a const-qualified type or has been declared with the readonly storage-class modifier. This is not valid. A typical example is assigning a value to a const variable.

User Action

Either remove the const qualifier from the object's type, remove the readonly storage-class modifier from the object declaration, or rework the code so that the object is not written to.

NEEDNONVOID, *context*"*expression*" has void type, but occurs in a context that requires a non-void result.

Description

An expression that must not be a void type was void. For example, the control expression for an if statement must not have void type.

User Action

Modify the expression so that it has the required type.

NEEDPOINTER, *context*"*expression*" has *type* type, but occurs in a context that requires a pointer.

Description

An expression that must be a pointer type was not a pointer type. For example, the operand of the dereference operator must be a pointer type.

User Action

Modify the expression so that it has a pointer type.

NEEDPTROBJ, *context*"*expression*" does not point to an object type.

Description

An expression that must be a pointer to an object type is a pointer to a function or incomplete type. For example, if a pointer is the operand of the postincrement operator, it must point to an object type.

User Action

Modify the expression so that it is a pointer to an object type.

NEEDSCALAR, *context*"*expression*" has *type* type, which is not scalar.

Description

An expression that must be a scalar type was not scalar. For example, only scalars can be cast to other types.

User Action

Modify the expression so that it is a scalar type.

NEEDSCALARTYP, *context*"*source type*" is *type* type, which is not scalar.

Description

In a cast expression, the destination type of the cast is not a scalar type. This is not valid. Both the source and target type of a cast must be scalars.

User Action

Modify the cast destination type so that it is a scalar type.

NEEDSIMPLEASM, This asm is unsupported or illegal.

Description

The argument to the asm intrinsic is invalid.

User Action

Supply a valid argument to the asm instrinsic.

NEEDSTRCONST, *context*"*name*" is not a legal asm string, a string constant is required.

Description

The argument to the asm intrinsic must be a string constant.

User Action

Change the argument to be a string constant.

NEEDSTRUCT, *context"expression"* has *type* type, but occurs in a context that requires a union or struct.

Description

The left operand of the . or -> operator does not have struct or union type.

User Action

Correct the operand.

NEGATIVEHINT, *context*a negative hint value is not allowed. The hint will be ignored.

Description

All #pragma hint directives must take positive floating point values.

User Action

Correct the hint.

NESTEDCOMMENT, Opening comment delimiter found inside a delimited comment; a previous comment may be missing its closing delimiter.

Description

C comments delimited by /* */ do not nest. When /* is encountered inside a delimited comment it usually means that the previous comment is missing its terminating */ or that the user has ill-advisedly attempted to "comment out" a section of code that contains a delimited comment.

User Action

It is traditional in C to use #if 0 to conditionalize out large sections of code. You may also want to consider //-style comments if the compiler modes you care about recognize them.

NESTEDENUM, The type "*type*" is declared nested within "*enclosing type*". In C, the nesting is ignored and *type* and its enumerator constants can be accessed as if they were not nested. *However, the type and its enumerators are members in C++. Fix*.

Description

C allows types to be declared within other types. For example: struct S { int a; enum E { first, second, third} b; int c; }; In C++ the enum E would not be accessible without using the :: operator.

User Action

Declare the nested type before declaring the enclosing type.

NESTEDTYPE, The type "*type*" is declared nested within "*enclosing type*". In C, the nesting is ignored and *type* can be accessed as if it were not nested. *However, the type is a member in C++. Fix*.

Description

C allows types to be declared within other types. For example: struct S { int x; struct S1 { int a; int b; } y; }; In C++ the struct S1 would not be accessible without using the :: operator.

User Action

Declare the nested type before declaring the enclosing type.

NESTINCL, Files included by this file are referenced. However nothing else appears to be referenced from this file.

Description

When compiling with the current set of compilation options, to improve compilation efficiency, you may wish to include the files which this file includes directly, rather than including them from this file.

User Action

For compilation efficiency, you may exclude this include file when compiling with the current set of compilation options.

NEWLOCALE, The compiler could not set its locale to the locale-specific native environment. This problem might be caused by an incorrect value for a name defined in your process environment such as "LC_ALL" or "LANG". The "C" locale will be used.

Description

During start-up, the compiler was unable to set its locale to the locale-specific environment. As part of its initialization, the compiler will issue the call setlocale(LC_ALL, ""). If this call fails, the compiler will set its locale to the "C" locale. In general, this message is output because the locale-specific native environment has been set incorrectly.

User Action

The best way to determine why the compiler was unable to set the locale is to write a small program that contains the library call setlocale(LC_ALL, "") and then examine the return value from the call.

NLCHAR, An unexpected newline character is present in a character constant.

Description

An end of line was encountered during the scanning of a character constant.

User Action

Terminate the character constant with a closing single quote character before the end of line.

NLHEADER, A newline occurs inside of a header name.

Description

An end of line was encountered before the closing double quote or angle bracket of an #include directive.

User Action

Terminate the directive argument properly.

NLSTRING, An unexpected newline character is present in a string literal.

Description

An end of line was encountered during the scanning of a string literal.

User Action

Terminate the string constant with a closing double quote character before the end of line, or continue the line with a continuation character.

NOADD, *context*"*expression1*" and "*expression2*" cannot be added.

Description

Because of their types, the two expressions cannot be used as the operands of the addition operator. Either both operands must be arithmetic type, or one operand must be a pointer to an object type and the other must be an integral type.

User Action

Modify the addition to use valid types.

NOBIFDISABLE, The function "*routine name*" is a builtin function reserved to the compiler, and cannot be used with #pragma function. The function will continue to be treated as a builtin.

Description

A function identifier specified in a #pragma function directive is the name of a builtin function. These functions cannot be explicitly disabled, they are always handled as builtin functions.

User Action

Remove the inappropriate use of the pragma, and change the name of the function in order to have it treated as an ordinary callable function.

NOBITFIELD, *context*"*expression*" is a bitfield, but occurs in a context that precludes bitfields.

Description

An expression that must not be a bitfield was a bitfield. For example, the operand of the address-of operator must not be a bitfield.

User Action

Modify the expression so that its type is not a bitfield type.

NOCASEHERE, This case label occurs outside of any switch statement.

Description

A case label can only occur inside of a switch statement.

User Action

Remove the case label.

NOCDDHERE, CDD is not available on this platform. The #dictionary directive has been ignored.

Description

The #dictionary directive requires CDD to be present on the platform. This directive will only be recognized on OpenVMS systems.

User Action

Remove the directive.

NOCOLON, Missing ":".

Description

The compiler was expecting a colon, but one was not found.

User Action

Correct the program syntax.

NOCOLONINEXPR, Missing colon for conditional expression.

Description

A conditional expression that occurs as part of a preprocessing expression was missing the ":" that separates the second from the third operand. The value of the resulting expression is undefined.

User Action

Correct the conditional expression.

NOCOMMA, Missing ",".

Description

The compiler was expecting a comma, but one was not found.

User Action

Correct the program syntax.

NOCONDEXPR, Missing #if conditional expression.

Description

An argument was not supplied to an #if or #elif preprocessing directive. The missing argument will cause the compiler to consider these as FALSE conditionals.

User Action

Supply a valid argument to the directive.

NOCONVERT, *context*"*expression*" is of type "*type*", and cannot be converted to "*target type*".

Description

An expression of one type cannot be converted to the type required by this expression. This most often occurs when the source type of an assignment or cast cannot be converted to the destination type. The rules for which types can be converted are rather complicated and differ based upon the compiler mode. Refer to the language documentation for a complete list of valid combinations.

User Action

Modify the conversion to use valid types.

NOCONVERTCLS, *context*"*expression*" is of type "*type*", and cannot be converted to a different "*type*" type.

Description

A struct or union of one type cannot be converted to a different struct or union type.

User Action

Modify the conversion to use valid types.

NODCL, *context*nothing is declared.

Description

The C standard requires that a declaration must declare at least a tag, an enumeration constant, or a declarator. This declaration contains none of these. This might not have been what you intended.

User Action

Correct or remove the declaration.

NODEFAULTHERE, This default label occurs outside of any switch statement.

Description

A case default label can only occur inside of a switch statement.

User Action

Remove the case default label.

NOENDIF, Missing #endif directive.

Description

The compiler encountered an #if, #ifdef, or #ifndef preprocessing directive without a matching #endif. This might not have been what you intended. The compiler will add the necessary #endif directive at the end of the compilation unit.

User Action

Make sure every #if, #ifdef and #ifndef has a matching #endif.

NOEQUAL, Missing "=".

Description

The compiler was expecting to see an "=" after the secondary_name specification of a #pragma weak or #pragma external_name directive. This message is only output on UNIX.

User Action

Correct the #pragma directive.

NOEQUALITY, *context*"*expression1*" and "*expression2*" cannot be compared for equality or inequality.

Description

Because of their types, the two expressions cannot be compared for equality or inequality. The rules for which types can be compared are rather complicated and differ based upon the compiler mode. Refer to the language documentation for a complete list of valid combinations.

User Action

Modify the comparison to use valid types. This can often be done by casting one of the expressions to the type of the other.

NOEXCEPTFLTR, *context* this exception handling call is not within an exception filter of a try block.

Description

The exception handling call must appear within an exception filter of a try statement block.

User Action

Either remove the exception handling call, or place it in a try statement block.

NOFBDAT, *text* does not contain feedback data

Description

The file indicated by the -feedback switch exists, but does not contain feedback data. This is probably an error on the users part, although it might be seen as part of the bootstrapping process.

User Action

Create a valid feedback file

NOFBFIL, Feedback file *text* does not exist

Description

The file specified after the -feedback option does not exist. This is normal during the bootstrapping process.

User Action

Either correct the spelling of the feedback option, or create the required feedback file.

NOFBOPT, Compilation will proceed without feedback optimizations

Description

A condition has occurred that has prevented the compiler from using feedback optimizations. This message is most often preceded by another message that will provide additional information.

User Action

Correct the condition that prevented the feedback optimizations.

NOFBRTN, Feedback inactive for *text* in this compilation

Description

Feedback information has gone stale for a particular routine (the source for the routine has changed). Feedback optimizations will not be applied to this routine.

User Action

Create a new feedback file

NOFIFILE, Cannot find include file *filename* specified on the command line.

Description

The header file name specified in the UNIX -FI command line option or the OpenVMS / FIRST_INCLUDE qualifier was not found using the search rules in effect for the quoted form of #include directives.

User Action

Either change the name of the file following the option or create the file.

NOFNTPDEFDECL, There is no identifier named "*name*" declared as a function or function typedef in this compilation unit.

Description

A #pragma assert and/or #pragma linkage directive(s) contains an identifier that is not declared as a function or function typedef in the compilation unit. This may not have been what you intented.

User Action

Remove the identifier from the #pragma assert and/or #pragma linkage, or declare it as a function or function typedef. empty string.

NOFORMALPARM, Missing formal parameter specifier.

Description

While processing the formal parameter list of a macro definition, the compiler encountered a missing formal parameter specifier. The macro will be be defined and this parameter ignored, but that may not have been what you intended.

User Action

Correct the formal parameter list so that it consists of a comma separated list of identifiers.

NOFUNC, There is no function named *name* defined in this compilation unit.

Description

A function that appears in a #pragma weak and is not defined in the compilation unit.

User Action

Either define the function or remove the function name from the pragma.

NOFUNC1, There is no definition for the inline function named *name* in this compilation unit.

Description

A function that appears in a #pragma inline or #pragma noinline, or is declared with the __inline or __forceinline storage class modifier, is not defined in the compilation unit.

User Action

Either define the function or remove the function name from the pragma, or remove the storage class modifier or the function specifier from the declaration.

NOIDFOUND, *context*an identifier was expected but not found.

Description

The compiler was expecting an identifier, but one was not found.

User Action

Correct the program syntax.

NOIDINPACKPOP, pragma pack pop directive has no identifier *name* which was found on the top of the pack stack.

Description

The #pragma pack (pop) directive has no identifier specified while the top element of the pack stack has one. Either this #pragma pack pop should have the identifier found on the stack, or this is an extra pragma pack pop, or the identifier should not be pushed by the corresponding #pragma pack push.

User Action

Check whether the pragma pack pop should have the identifier. Verify that there's no extra #pragma pack pop or #pragma member_alignment restore which popped the identifier to the top of the pack stack. Correct the directive(s).

NOINCLFILE, Cannot find file *filename* specified in #include directive.

Description

The specified include file does not exist.

User Action

Either change the name of the file in the #include preprocessing directive, or create the include file.

NOINCLFILEF, Cannot find file *filename* specified in #include directive.

Description

The specified include file does not exist.

User Action

Either change the name of the file in the #include preprocessing directive, or create the include file.

NOINCLUDEARG, #include directive missing argument.

Description

An argument was not supplied to an #include preprocessing directive. The directive will be ignored.

User Action

Supply a valid argument to the directive.

NOINIT, The type of *variable* does not permit initialization.

Description

This type cannot be initialized. Only objects and arrays of unknown size can be initialized.

User Action

Remove the initializer.

NOINLFUNC, There is no definition for the inline function named *name* in this compilation unit.

Description

A function is declared with an __inline or inline keyword and is not defined in the compilation unit.

User Action

Either define the function or remove the __inline or inline keyword from the declaration.

NOINLINEM, The main function cannot be inlined.

Description

The C99 standard prohibits the inline keyword from being used on the main function.

User Action

Remove the inline keyword.

NOINLINEREF, *context* "*name*" has internal linkage and is referenced from an an inline auxiliary function. This is a violation of the C99 Standard.

Description

A function declared with the inline keyword and without a declaration containing the keyword, extern, or without a declaration which lacks the inline keyword and the static keyword declares an auxiliary inline declaration. A definition of an auxiliary inline shall not contain a definition of a modifiable object with static storage duration, and shall not contain a reference to an identifier with internal linkage.

User Action

Remove the inline keyword from all declarations of the parent function, or if it is appropriate, change the declaration of the referenced item to a declaration which has something other than internal linkage.

NOINLINEST, In an inline auxiliary function, the modifiable object "*name*" is declared with static storage duration. This is a violation of the C99 standard.

Description

A function declared with the inline keyword and without a declaration containing the keyword, extern, or without a declaration which lacks the inline keyword and the static keyword declares an auxiliary inline declaration. A definition of an auxiliary inline shall not contain a definition of a modifiable object with static storage duration, and shall not contain a reference to an identifier with internal linkage.

User Action

Remove the inline keyword from all declarations of the parent function, or if it is appropriate, add the const keyword to the declaration to create an non-modifiable object.

NOLEAVETARG, This leave statement is not within a try statement.

Description

The exception handling statement leave must appear within a try statement block.

User Action

Either remove the leave statement, or place it in a try statement block.

NOLEFTOPERND, Token pasting operator missing left operand.

Description

The preprocessing token pasting operator "##" appears in a macro body without the preceeding token argument.

User Action

Either remove the operator or supply it with two tokens that will be pasted together.

NOLINKAGE, *context*"*name*" has no linkage and has a prior declaration in this scope at *where*.

Description

A declaration within a function body redeclares an identifier declared earlier in the current scope, and both declarations did not have the extern storage class.

User Action

Either remove the extra declarations, or have all declarations for the identifier use the extern storage class.

NOMACRONAME, #define directive is missing macro name identifier.

Description

The #define preprocessing directive was not supplied with an argument. The directive should be followed with an identifier that specifies the macro name to be defined. The directive will be ignored.

User Action

Supply a valid argument to the preprocessing directive.

NOMAINUFLO, No main function encountered within module. / IEEE_MODE=UNDERFLOW_TO_ZERO is ignored.

Description

Use of the /IEEE_MODE=UNDERFLOW_TO_ZERO is only meaningful for compilation units that contain a main program. The compiler will ignore the qualifier.

User Action

Remove the qualifier from the command line.

NOMAPPOSSIBLE, The register "*register*" cannot be mapped to a register on the target platform.

Description

The pragma linkage directive contains architecture-specific information. The Alpha register conventions are different from the IA64 registers conventions. The compiler will normally try to map the Alpha registers to the corresponding registers on IA-64. In this case this register cannot be mapped because there is no corresponding IA64 register.

User Action

Update the linkage to use a register that can be mapped, or specify the linkage to use the linkage_ia64 directive.

NONAMEMEMBERS, *context*a struct or union has no named members. This is undefined behavior according to the C standard.

Description

The C standard requires that a struct or union contain at least one named member. Because this struct/union contains no named members, it does not conform to the C standard and might not be portable.

User Action

Make sure at least one member has a name.

NONATOMIC, Unable to generate code for atomic access

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

NONEWTYPE, The type "*type*" is being declared as part of *context*. C++ does not permit a new type to be declared in this context. *Fix*.

Description

C++ does not allow types to be declared in certain contexts that are valid in C. One example is the declaration of a type within a function prototype.

User Action

Declare the type before its use.

NONGRNACC, Unable to generate code for requested granularity

Description

The compiler has generated a call a routine that performs longword operations on some data that is requested to be accessed with byte granularity. Because of this, the requested granularity will not be met for this data access. This routine may be generated for a memory copy routine (such as memcpy). The call can also be generated for certain struct assignments.

User Action

If the data must be accessed with byte granularity then write your own routine that does the required action using byte objects. If byte granularity is not needed at this point, the message can be ignored.

NONINTENUM, *context*the enumeration type, and all associated enumeration constants will have type *type* because at least one enumeration constant had a value that could not be represented in the type signed int.

Description

The standard requires that enumeration constants have a value representable as an int. Other C compilers will allow enumeration constants to have values outside this range. In some modes the VSI C compiler will allow this extension. To identify exactly which constants are outside the range, enable the nonintenumcon message.

User Action

Be aware that other compilers may not support this extension.

NONINTENUMCON, *context*allowing an enumeration constant outside the range of signed int is a language extension.

Description

The standard requires that enumeration constants have a value representable as an int. Other C compilers will allow enumeration constants to have values outside the range. In some modes the VSI C compiler will allow this extension.

User Action

Be aware that other compilers may not support this extension.

NONINTENUMCON1, *context*this enumeration constant and its associated enumeration type will not have the type signed int. This behavior differs from earlier versions of the compiler.

Description

This message will only be output when the "enumrange" message is disabled. This version of the VSI C compiler will allow enum constants to have a type other than signed int. This is for compatibility with other compilers. Programs that rely on the compiler to truncate enum constants may not work as expected. For more information, enable the nonintenum message.

User Action

If your program relies on this truncation, cast the constant to int.

NONLBEFOREEOF, File does not end in unescaped newline.

Description

The final character of a file was not a newline character. This could indicate that the file has been corrupted. The compiler will insert a newline character at this point in the input stream.

User Action

Update the source file so that it ends with a newline.

NONMULTALIGN, The size of this structure is *size* bytes, which is not a multiple of its alignment of *align*. Respecify the alignment of the structure or add *bytes* bytes of additional padding.

Description

The size of a struct or union is not a multiple of its alignment. This could cause unaligned accesses if an array of these structs or unions is declared.

User Action

Modify the struct/union or the alignment so that the size of the struct or union is a multiple of the alignment.

NONOCTAL, An octal constant contains non-octal digits.

Description

An octal constant contains a non-octal digit. The compiler will convert this non-octal digit to its corresponding octal value and use that value instead. For example, 0190 will be converted to 0210 (decimal 136) as the non-octal digit 9 is converted to the octal 11.

User Action

Correct the octal constant to use only octal digits.

NONPORTDEFINED, "defined" is treated as an identifier here, not an operator.

Description

For compatibility with older C compilers, in certain modes the compiler will treat #ifdef defined(foo) as #ifdef defined, and #ifndef defined(foo) as #ifndef defined. This might not have been what you intended.

User Action

Do not mix #ifdef/#ifndef with the defined operator.

NONPORTLINEDIR, Non-standard #line directive.

Description

Accepting the line directive without the "line" preprocessing keyword is an extension of VSI C. The program does not conform to the C standard, and might not be accepted by other compilers.

User Action

Add the "line" preprocessing keyword to the directive.

NONSTANDCAST, *context*"*expression*" of type "*type*", is being converted to "*target type*". Such a cast is not permitted by the standard.

Description

The standard only permits casts from a pointer to an object incomplete type to another pointer to an object or incomplete type, or from a pointer to function type to another pointer to function type. Note that void is considered an incomplete type, so casts between pointer to void and pointer to function types are not permitted by the C standard.

User Action

Be aware of this difference if you plan to port this source to another compiler.

NONULINIT, *context*, there is no room for the terminating '\0'. Standard C allows this, but C++ does not.

Description

This declaration initializes an object to a strict literal. Although the object is large enough to hold the characters in the literal, it is not large enough to hold the terminating null character. This might not have been what you intended. This practice is also not valid in C++.

User Action

Increase the size of the object, or reduce the size of the initializer.

NOOPERAND, Stringization operator missing operand.

Description

The preprocessing stringization operator "#" appears in a macro body without a token argument after the operator.

User Action

Either remove the operator or supply it with a token that will be stringized.

NOOPERANDS, Token pasting operator missing both operands.

Description

The preprocessing token pasting operator "##" appears in a macro body without either the preceeding or following token arguments.

User Action

Either remove the operator or supply it with two tokens that will be pasted together.

NOPARENARGLST, Missing right parenthesis for macro argument list.

Description

A macro invocation's argument list did not end in a right parenthesis.

User Action

Correct the program syntax.

NOPARM, This declaration does not declare a parameter.

Description

The parameter declaration list of an old-style function definition included a type but no parameter identifier.

User Action

Replace the old-style function definition with the recommended prototype-format declaration. If this is not possible, include the correct identifier after the parameter type.

NOPARMLIST, The declaration of *function* has an empty parameter list. If the function has parameters, they should be declared here; if it has no parameters, "void" should be specified in the parameter list.

Description

The recommended way to declare a function that takes no parameters is to use "void" in the parameter list.

User Action

Make the recommended change.

NOPRAGARG, No argument for #pragma *pragma* was found. Pragma is ignored.

Description

A #pragma directive was not followed by one of the expected arguments. The directive will be ignored.

User Action

Supply all required arguments to the directive.

NOPSECT, Missing psect name.

Description

The psect specifier in a globaldef declaration must be a string constant.

User Action

Either make the psect a string constant or remove the psect specifier.

NOREGAVAIL, Unable to satisfy program register allocation requirements.

Description

The compiler is unable to allocate all the registers requested by the program. This most often happens when asm directives require too many registers.

User Action

Rework the asm directives so they use fewer registers

NORELATIONAL, *context*"*expression1*" and "*expression2*" cannot be compared with a relational operator.

Description

Because of their types, the two expressions cannot be used as the operands of a relational operator. The rules for which types can be used in a relational are rather complicated and differ based upon the compiler mode. Refer to the language documentation for a complete list of valid combinations.

User Action

Modify the relational to use valid types. This can often be done by casting one of the expressions to the type of the other.

NORETNONVOID, noreturn assertion of #pragma assert directive can't be specified for non-void function.

Description

noreturn assertion was specified in #pragma assert directive for non-void function; the noreturn assertion will be ignored.

User Action

Either remove noreturn assertion from the directive, or change return type for the function to void.

NORETURNVAL, The function "*name*" returns a value, but no value is given in this return statement.

Description

A function that returns a value contains a return statement that is missing a return value. Therefore, the returned value will be undefined. This might not have been what you intended.

User Action

Supply a return value for the return statement.

NORETURNVAL1, The function "*name*" has an implicit return type of int, but no value is given in this return statement.

Description

A function that has an implicit return type of int contains a return statement that is missing a return value. Therefore, the returned value will be undefined. This might not have been what you intended.

User Action

Supply a return value for the return statement or define the function with a void return type.

NORETVAL, routine *text* does not return a value

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

NORGHTPAREN, No right parenthesis for parameter list.

Description

A macro definition's formal parameter list did not end in a right parenthesis. A right parenthesis has been assumed.

User Action

End the formal parameter list with a right parenthesis.

NORIGHTOPERND, Token pasting operator missing right operand.

Description

The preprocessing token pasting operator "##" appears in a macro body without a token argument after the operator.

User Action

Either remove the operator or supply it with two tokens that will be pasted together.

NORIGHTPAREN, Missing ")".

Description

A right parenthesis was expected at this point in the program, but none was found.

User Action

Correct the program syntax.

NOSEHHAND, Missing exception handler.

Description

The __builtin_try clause must specify an exception handler of either __builtin_finally or __builtin_except. This message is only generated on UNIX systems.

User Action

Correct the __builtin_try clause.

NOSEMI, Missing ";".

Description

The compiler was expecting a semicolon, but one was not found.

User Action

Correct the program syntax.

NOSEMI1, Missing ";". This condition may have been caused by an open brace without a matching close brace. The compiler will attempt to identify open braces that might be missing a close brace.

Description

The compiler was expecting a semicolon, but one was not found. This condition may have been caused missing close brace. This message is followed by some number of additional messages that attempt to identify the bad open brace.

User Action

Correct the program syntax.

NOSEMISTRUCT, Missing ";" after last structure or union member.

Description

Accepting a struct/union type without a semicolon after the last member specifier is a language extension of VSI C provided for compatiblity with older C compilers. This syntax is not valid in standard C, and may not be accepted by other C compilers.

User Action

Add the semicolon at the end of the last member.

NOSFILE, Cannot create .s file: overlapping static storage initializations at Psect *text + number*

Description

When producing an output assembly file, the compiler back-end as detected a case where the same storage location has been initialized to more than one value. This can occur when inter-file optimization has been enabled.

User Action

Remove one of the initializers.

NOSHAREEXT, noshare is a language extension.

Description

The noshare storage class modifier is a language extension of VSI C. Other C compilers might not successfully compile a program that uses the extension.

User Action

Be aware of this extension if you wish to port the code.

NOSHRINSHR, Noshare variable resides in shr extern model - noshare ignored.

Description

The current extern model places all external objects in a shareable section. Placing an object with a noshare type qualifier in such a section is invalid. The compiler will ignore the noshare type qualifier

User Action

Place noshare objects in sections with the noshare attribute.

NOSTRING, Missing string literal.

Description

The compiler was expecting a string literal, but one was not found.

User Action

Correct the program syntax.

NOSUBTRACT, *context*"*expression2*" cannot be subtracted from "*expression1*".

Description

Because of their types, the two expressions cannot be used as the operands of the subtraction operator. Either both operands must be arithmetic type, or both operands must be pointers to qualified or unqualified versions of compatible object types, or the left operand must be a pointer type and the right operand must be an integral type.

User Action

Modify the subtraction to use valid types.

NOTADDRCAST, *context*the address constant "*expression*" can be cast only to a pointer type, but "*type*" is *type class* type.

Description

An address constant can only be cast to a pointer type.

User Action

Correct the cast.

NOTAREDUCTION, bad reduction path from fetch of *text*

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

NOTCOMPAT, *context*the type of "*name*" is not compatible with the type of a previous declaration of "*name*" at *location*.

Description

The same identifier has been redeclared with a type that is incompatible with the type given in a previous visible declaration. In some modes, the compiler will use the redeclared type as the type of the identifier.

User Action

Change the declarations to use compatible types.

NOTCOMPATIMP, *context*the type of the function "*name*" is not compatible with the earlier implicit declaration of "*name*" at *location*.

Description

The declared type of a function does not match the type given to the function by its earlier implicit declaration. This may cause unexpected behavior.

User Action

Make sure that a valid function declaration is visible at the point in the source where the function is first called.

NOTCOMPFUNC, *context* an invalid redeclaration of "*name*" to or from a function type is being ignored.

Description

In certain modes, the compiler will allow an identifier to be redeclared with a different type. In this case, the type of the redeclaration is used. However, in cases where the identifier is redeclared to or from a function type, the redeclaration is ignored.

User Action

Remove the redeclaration of the identifier.

NOTCONSTQUAL, *context*the referenced type of the pointer value "*expression*" is const, but the referenced type of the target of this assignment is not.

Description

In an assignment of two pointer types, the type pointed to by the destination operand must have all the type qualifiers of the type pointed to by the source operand. In this case, the type pointed to by the source has the const type qualifier, but the type pointed to by the destination does not.

User Action

Correct the assignment to use compatible types. This can be done by inserting a cast operand.

NOTEXPECTING, Error parsing *what*. Found "*found*" when expecting *expecting*.

Description

While parsing the program, the compiler has encountered something unexpected. The message will detail what the compiler was trying to parse and the item that was invalid, and will also produce a list of those items it was expecting to find.

User Action

Correct the offending section of the program.

NOTINCRTL, Identifier "*id*" is reserved by the *C89|C99|C2010...* standard and will be mapped to "*name*" although it is not available in the CRTL available to the compiler.

Description

The specified identifier is reserved for use as an identifer with external lingage in the specified version of the C standard. But according to the CRTL mapping table available to the compiler, that identifier is not defined in the CRTL you expect to link against. This may be because the function or object is not yet implemented in the current DECC$SHR, or because you have used logical DECC$CRTLMAP to specify a CRTL mapping table for a version of the CRTL that does not implement it.

User Action

If you intended to use the identifier as defined by the C standard, and you have not defined the logical DECC$CRTLMAP, then the identifier is not defined in the DECC$SHR available to the compiler. If this is the latest released DECC$SHR, then the identifier is not yet implemented and you need to consider workarounds; otherwise you should upgrade to the latest available CRTL that does implement it. If you did not intend to use the identifier as defined by the C standard (i.e. it is an identifier you expected to be defined by your application), then you have a name clash with the specified version of the standard and you should change the spelling of the identifier; alternatively, you could disable prefixing for it using /PREFIX=EXCEPT=, or specify an older version of the standard with either /PREFIX= or /STANDARD=.

NOTINTRINSIC, The function "*routine name*" is not a known intrinsic function and cannot be used with #pragma function. The function is unaffected by this pragma.

Description

A function identifier specified in a #pragma function directive is not a valid intrinsic function on this platform. The function is thus never treated as an intrinsic, and so #pragma function can never be applicable to it. Perhaps the name was misspelled, or perhaps the function was thought to be intrinsic, possibly because it is intrinsic on some other platform. In the latter case, the desired result, that the function not be treated as intrinsic, would happen with or without the pragma.

User Action

Either correct the identifier spelling or remove the use of the pragma.

NOTLOCALPARM, *context*"*identifier*" is not a local parameter.

Description

The second argument to the variable argument list va_start macro is not a formal parameter of the current function. The second argument to va_start should be the rightmost parameter in the function definition.

User Action

Correct the second argument to va_start.

NOTONEORZERO, *context*the value of "*expression*" is neither 0 nor 1.

Description

The __builtin_va_start macro has been used incorrectly.

User Action

Correct the use of the macro.

NOTPARM, *contextname* is not a parameter.

Description

The identifier name in the parameter declaration does not match a name in the identifier list of an old-style function definition.

User Action

Correct either the identifier in the declaration or in the identifier list so that they match. VSI also recommends that old-style function definitions be replaced by prototype-format definitions.

NOTPOSINT, *context*the array bound "*expression*" is not a positive integer.

Description

The compiler has encountered an array-bounds specifier that is either zero or negative. Array-bounds specifiers must be positive integer constants.

User Action

Correct the array-bounds specifier

NOTRESTQUAL, *context*the referenced type of the pointer value "*expression*" is restrict, but the referenced type of the target of this assignment is not.

Description

In an assignment of two pointer types, the type pointed to by the destination operand must have all the type qualifiers of the type pointed to by the source operand. In this case, the type pointed to by the source has the restrict type qualifier, but the type pointed to by the destination does not.

User Action

Correct the assignment to use compatible types. This can be done by inserting a cast operand. Note that care should be taken in assigning to a restricted pointer type.

NOTRIGHTMOST, *context*"*identifier*" is not the rightmost parameter to "*function*".

Description

The second argument to va_start was not the rightmost parameter in the variable parameter list in the function definition. This is an invalid argument to va_start. Other compilers might not accept this program.

User Action

Update the second argument to va_start to use the rightmost parameter.

NOTSCALARCTRL, The controlling expression "*expression*" has *type* type, which is not scalar.

Description

An execution control expression does not have scalar type. This is not valid. An example of an execution control expression is the expression following the while keyword in a while statement.

User Action

Change the control expression to have scalar type.

NOTTYPEDEF, *context*"*name*" does not name a type.

Description

This message is output when the compiler encounters an identifier that it believes is a typedef and no valid typedef by this name is defined in the current scope. This most often occurs when there was an error in the declaration of the typedef name.

User Action

Correct the declaration of typedef.

NOTUNALQUA, *context*the referenced type of the pointer value "*expression*" is __unaligned, but the referenced type of the target of this assignment is not.

Description

In an assignment of two pointer types, the type pointed to by the destination operand must have all the type qualifiers of the type pointed to by the source operand. In this case, the type pointed to by the source has the __unaligned type qualifier, but the type pointed to by the destination does not.

User Action

Correct the assignment to use compatible types. This can be done by inserting a cast operand.

NOTVOLQUAL, *context*the referenced type of the pointer value "*expression*" is volatile, but the referenced type of the target of this assignment is not.

Description

In an assignment of two pointer types, the type pointed to by the destination operand must have all the type qualifiers of the type pointed to by the source operand. In this case, the type pointed to by the source has the volatile type qualifier, but the type pointed to by the destination does not.

User Action

Correct the assignment to use compatible types. This can be done by inserting a cast operand.

NOTYPES, Declaration has no type or storage class.

Description

A file-scope declaration contains no type and no storage-class specifier. In some modes, the VSI C compiler will treat this as a tentative definition of an int variable. Accepting this declaration is an extension to standard C provided for compatibility with other compilers.

User Action

Rewrite the declaration to contain a data type and/or storage class.

NOUNIQFORMALS, Non-unique formal parameter definition.

Description

The same name has been used for more than one formal parameter in a macro definition. Any occurrence of the name in the macro body will correspond to the last formal parameter given this name.

User Action

Each macro formal parameter should have a unique name.

NOWHILE, Missing "while".

Description

While processing a do statement, the compiler did not find a while clause.

User Action

Supply a while clause for the do statement.

NOWRTWRITTEN, Readonly psect *text* is written

Description

The compiler has detected an attempt to write to read-only storage.

User Action

Either remove the write or make the storage read/write.

OBJECTTOOBIG, The size of "*name*" exceeds the maximum size of an object allowed on this platform which is *size* bytes.

Description

An object has been declared with a size that is too large for this platform.

User Action

Reduce the size of the object.

OKCPPINARGS, "*#directive*" directive within a macro argument list is not portable.

Description

VSI C will allow certain directives to appear within the argument list of a macro invocation. This might not be portable.

User Action

If possible, rewrite the macro invocation.

OPENBRACE, Missing "{".

Description

The compiler was expecting an open brace, but one was not found.

User Action

Correct the program syntax.

OPENCOMMENT, A comment is not terminated.

Description

The end of a file was reached while within a comment. The message will indicate the start of the comment. All source files, even those included via the #include preprocessing directive, must not end in a pending comment.

User Action

Terminate the comment before the end of the source file.

OPENPAREN, Missing "(".

Description

The compiler was expecting an open parenthesis, but one was not found.

User Action

Correct the program syntax.

OPTIMIZEPOP, This "restore" has underflowed the pragma optimize stack. No corresponding "save" was found.

Description

The optimize stack, managed by the #pragma optimize and #pragma environment directives, contains more restores than saves. This could signify a coding or logic error in the program.

User Action

Make sure each restore has a corresponding save.

OPTLEVEL, Invalid optimization level *number*, defaulted to *number*.

Description

An optimization level that is outside the range of valid optimization levels has been specified. The compiler will default to the stated level.

User Action

Supply a valid optimization level on the command line.

OTHERDECLUSED, *context*"*name*" is not declared in a scope active at this point in the compilation. However, there is a declaration of this identifer with extern storage class in another scope at *where*. This declaration will be used.

Description

In some modes, if the compiler cannot find the declaration of an object in the current scope, it will search other scopes for extern declarations of that object. If it finds such a declaration, it will be used. Note that this is a language extension provided for compatibility with other compilers.

User Action

Declare the object so that it is visible at all places it is referenced.

OTHERMEMBER, *context*"*name*" is a member of another struct or union.

Description

In certain modes, the compiler will allow a struct or union reference whose right operand is not a member of the struct or union type of the left operand. This is allowed for compatibility with other compilers.

User Action

Correct the struct or union reference so that the member specifier is a member of the type of the left operand.

OUTARGPREC, *context*the type of this argument to *function name* is not appropriate for the precision argument of the conversion specifier "*incorrect conversion*". Behavior can be unpredictable.

Description

This argument corresponds to an output precision specification. C requires that this argument have integer type, and it does not.

User Action

Cast the argument to an int type.

OUTARGWIDTH, *context*the type of this argument to *function name* is not appropriate for the width argument of the conversion specifier "*incorrect conversion*". Behavior can be unpredictable.

Description

This argument corresponds to an output width specifier. C requires that this argument have integer type, and it does not.

User Action

Cast the argument to an int type.

OUTFLOATINT, *context*this argument to *function name* and conversion specifier "*incorrect conversion*" combine integer and floating-point types. Behavior can be unpredictable.

Description

The compiler has detected an output conversion specifier whose data type does not match its corresponding argument in a way that will cause unpredictable behavior.

User Action

Modify either the argument or the conversion specifier so that they match.

OUTSTRINGTYPE, *context*this argument to *function name* is of "*type name*" type and is not appropriate for the conversion specifier "*incorrect conversion*". The value will be formatted in an unintended manner.

Description

The compiler has detected a string conversion specifier that does not match its corresponding argument. This might not have been what you intended.

User Action

Modify either the argument or the conversion specifier so that they match.

OUTTOOFEW, *context*the number of conversion specifiers to *function name* exceeds the number of values to be converted. Conversion specifiers from "*last valid conversion*" onward will process meaningless and perhaps invalid data.

Description

The number of conversion specifiers is greater than the number of values to be converted as specified in the parameter list. This is probably not what you intended.

User Action

Make sure the number of conversion specifiers match the values to be converted.

OUTTOOMANY, *context*additional arguments to *function name* are provided for which there are no conversion specifiers in the format string. Arguments from "*last expression*" onward will be evaluated, but not processed by *function name*.

Description

The number of conversion specifiers is less than the number of values to be converted as specified in the parameter list. This is probably not what you intended.

User Action

Make sure the number of conversion specifiers match the values to be converted.

OUTTYPELEN, *context*this argument to *function name* is of "*typeclass*" type and is not appropriate for the conversion specifier "*incorrect conversion*". The value might be truncated or formatted in an unintended manner.

Description

The compiler has detected an output conversion specifier that does not match its corresponding argument. This might not have been what you intended.

User Action

Modify either the argument or the conversion specifier so that they match.

OUTVARORDER, *context*variable ordering is used in a conversion specifier for *function name*. If variable ordering is used, it must be specified for all conversions.

Description

A conversion specification can contain only one type of conversion specification - % or %n$. Mixing them will cause unpredictable behavior.

User Action

Change the format specification to use only one type of conversion specification.

PACKSTACKPOP, This "pop" has underflowed the pragma *stack name* stack. No corresponding "push" was found.

Description

The member_alignment/pack stack, managed by the #pragma pack and #pragma member_alignment directives, contains more pops/restores than pushes/saves, This could signify a coding or logic error in the program.

User Action

Make sure each pop/restore has a corresponding push/save.

PARAMREDECL, *context*"*name*" overrides a formal parameter declared at *where*.

Description

A declaration within a function body redeclares a formal parameter.

User Action

Change the name of either the formal parameter or the declared variable.

PARENLITERAL, *context*accepting a string literal in parentheses as the initializer for a character array is a language extension.

Description

The compiler accepts this kind of initializer for compatibility with many other C compilers. According to the C standard, a string literal in parentheses is a character pointer. Therefore, this program does not conform to the standard and may be rejected by other compilers.

User Action

Remove the parentheses.

PARMINCOMP, *context*the parameter *name* has an incomplete type.

Description

The parameter of an old-style function definition has an incomplete type. This is not valid.

User Action

Complete the type before the declaration of the parameter. VSI also recommends that old-style function definitions be replaced by prototype-format definitions.

PARMINIT, *context*a parameter declaration cannot include an initializer.

Description

The parameter declaration list of an old-style function definition included an initializer. This is not valid.

User Action

Remove the initializer from the declaration and initialize the parameter in the function body. VSI also recommends that old-style function definitions be replaced by prototype-format definitions.

PARMSTORCLS, *context*a parameter has an explicit storage class other than "register".

Description

The only storage class that can be specified for a formal parameter is "register".

User Action

Either remove the storage class or use "register" if that is desired.

PARMSTORMOD, *context*a parameter cannot have a storage class modifier.

Description

A formal parameter cannot be declared with a storage class modifier.

User Action

Remove the storage class modifier.

PARMTYPLIST, Ill-formed parameter type list.

Description

While processing a function declaration, an invalid parameter type list was encountered.

User Action

Correct the program syntax.

PARNOIDENT, Missing identifier.

Description

While processing an old-style function definition, the compiler was expecting an identifier, but one was not found.

User Action

Correct the program syntax. VSI also recommends that old-style function definitions be replaced by prototype-format definitions.

PDBOPERR, Error opening PDB file *text*: *text*

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

PDBTYPERR, Error adding type record to PDB file: *text*

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

PDOINDEXNOTPRIV, index variable of PDO *text* is not a private variable

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

PDONEINSTATIC, pdone *text* in statically-scheduled PDO will be ignored

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

PDONENOTINPDO, pdone *text* is not nested in a PDO

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

PLUSWSTOCLS, The use of the *spelling* option has prevented this redeclaration of "*variable*" from changing its linkage. The linkage will be that specified by the earlier declaration at *location*.

Description

In many cases, the compiler will allow a redeclaration of an item to change its linkage. For example, in most modes, if an object is declared with extern linkage and later with static linkage, the compiler will give it static linkage. This changing of linkage usually causes a warning to be issued. However, in cases where interfile optimization has been selected (-ifo on UNIX, /PLUS_LIST_OPTIMIZE on OpenVMS), the compiler cannot allow a later declaration to modify the linkage of a previous declaration.

User Action

Change all declarations to use the same linkage.

POINTERINTCAST, *context*the 64-bit pointer "*expression*" is being cast to an integer type that is only *size* bits in size. This behavior is undefined.

Description

Casting a 64-bit pointer to a shorter integer type is undefined behavior. This also could indicate code that relies on pointers and integers being the same size. The code will cause an unexpected loss of data on 64-bit platforms.

User Action

If this is the intended behavior, first cast the pointer to a 64-bit integer, then cast the result to the desired integer type.

POPMISMATCH, The member alignment popped/restored with pragma *pragma name* was saved using pragma *pragma name*. The member alignment restored will take effect.

Description

VSI C supports two forms of the member alignment directives. One begins with #pragma pack, the other with #pragma member_alignment. A program has mixed the pack and the member_alignment form of the directives in a way that is not recommended. This might indicate a programming error.

User Action

If a member alignment has been saved by one form of the member-alignment directive, it should be restored by the same form of the directive.

PRAGIGNORE, The pointer size control *name* pragma is not active. Pragma is ignored.

Description

Either one of the pragmas that used to control pointer size has been specified on a platform that does not support mixed pointer sizes, or the #pragma pointer_size directive has been used without the appropriate command-line option or qualifier. In all cases, the directive is ignored.

User Action

Either remove the directive or add the appropriate command-line option.

PRAGMA, Strict standard C extension: A #pragma directive was encountered.

Description

As the purpose of a #pragma directive is to specify implementation-defined behavior, it is likely that other C compilers will not treat this pragma in the same way VSI C will.

User Action

Be aware of this if you wish to port the program.

PRAGMAIDENT, Please use the preferred "#pragma ident" directive in place of the "#ident" directive.

Description

The #ident directive is a language extension. Other C compilers might not accept it.

User Action

Use the portable #pragma ident directive instead.

PRAGMAINBLK, The pragma *name* cannot be used inside a function block.

Description

This #pragma directive is only permitted at file scope, outside of all function definitions.

User Action

Move the directive to file scope, preceding the function definition that is to be affected. To limit the pragma to just that particular function, sandwich the #pragma and the function definition between a pair of matching pragmas with the save and restore keywords.

PRAGMAMOD, Please use the preferred "#pragma module" directive in place of the "#module" directive.

Description

The #module directive is a language extension. Other C compilers are unlikely to accept it.

User Action

Use the portable #pragma module directive instead.

PRAGMAOPTDUP, This #pragma optimize has already modified this optimization setting. This setting will replace the old.

Description

A #pragma optimize has specified the same optimization setting more than once. The later setting will replace the previous one.

User Action

Remove the earlier setting.

PRAGMAOPTLVL, The level set by a #pragma optimize directive must be between 0 and 5. Pragma is ignored.

Description

A #pragma optimize has tried to set the optimization level to a value outside the valid range. The compiler will ignore the directive.

User Action

Set the optimization level to a number from 0 to 5.

PRAGMAOPTSPEC, Setting speculation control is not available on this platform. The setting will be ignored.

Description

Setting speculation control in only available on certain platforms. Trying to modify the setting on other platforms will have no effect.

User Action

Remove the speculation setting.

PRAGMAOPTZERO, If a #pragma optimize specifies level=0, that must be the only optimization setting specified by the pragma. Pragma is ignored.

Description

If a #pragma optimize specifies level=0, that must be the only optimization setting specified by the pragma. The compiler will ignore the directive.

User Action

Remove the other settings specified by the directive.

PREOPTE, An error was detected in the processing of a *option spelling* option: *#define or #undefine problem*

Description

An error was encountered during the processing of a macro definition specified on the command line. The message should provide additional information about the error.

User Action

Correct the command line argument.

PREOPTW, A problem was detected in the processing of a *option spelling* option: *#define or #undefine problem*

Description

A problem was encountered during the processing of a macro definition specified on the command line. The message should provide additional information about the problem.

User Action

Correct the command-line argument.

PREPROCOUT, An error occurred while attempting to open either the preprocessor output file or the dependency file: *problem*.

Description

An unexpected error occurred during the creation of a preprocessor output file or a dependency file. The message text will contain additional information about the failure.

User Action

Correct the condition that caused the failure.

PRIVATENOTSHARE, variable *text* on a local or lastlocal list is not declared in a shared scope.

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

PROMOTMATCH, *context*the promoted type of *name* is incompatible with the type of the corresponding parameter in a prior declaration.

Description

The promoted type of a parameter of an old-style function declaration does not match the type given earlier in a prototype declaration of the function.

User Action

Correct the data types so they match. VSI also recommends that old-style function definitions be replaced by prototype-format definitions.

PROMOTMATCHW, *context*the promoted type of *name* is incompatible with the type of the corresponding parameter in a prior declaration.

Description

The promoted interger or floating type of a parameter of an old-style function declaration does not match the integer or floating type given earlier in a prototype declaration of the function.

User Action

Correct the data types so they match. VSI also recommends that old-style function definitions be replaced by prototype-format definitions.

PROTOF, An error occurred while attempting to open the prototype output file: *problem*.

Description

An unexpected error occurred during the creation of a prototype output file. The message text will contain additional information about the failure.

User Action

Correct the condition that caused the failure.

PROTOSCOPE, The type "*type*" has been declared within and is limited to a function prototype scope. It will not be compatible with an identical type declared in another scope. This might not be what you intended.

Description

A type is declared within a function prototype. The type is local to the function prototype and will not be visible outside the prototype. This might cause unexpected errors later in the compilation.

User Action

Declare the type before the function prototype.

PROTOSCOPE2, *context*the struct type was previously declared with prototype scope in this function. Now it is declared with a different prototype scope.

Description

This function declaration contains a parameter that is a pointer to a type that has prototype scope, and an earlier declaration of the function contains a parameter that is also a pointer to a type that has a different prototype scope. In most compiler modes this will cause the function redeclarations to differ.

User Action

Avoid declaring types with function prototype scope.

PROTOSCOPE3, *context*the struct type was previously declared in this function with prototype scope. Now it is declared with file scope.

Description

This message is generated when the compiler first encounters a function prototype that declares a type with prototype scope, and then later sees a second declaration or definition of that same function with the parameter declared using the same type declared at file scope. For example: void foo(struct S { int a; int b;} *s); struct S { int a; int b;} s; void foo(struct S *s);

User Action

Declare the type at file scope before the first prototype declaration.

PROTOSTATIC, The extracted header file contains prototypes for static functions, which should be removed before including the header in a source file other than the originator.

Description

When extracting function prototype declarations, the compiler has encountered a static function. The prototype declaration placed in the output .H file should be removed if the .H file is included in any source other than that used to create the .H file. This is because those static functions may not be declared in the other files. This message can only be generated when the compiler has been invoked with the option to extract function prototype declarations, and the suboption to generate prototypes for static functions has also been specified.

User Action

Be aware of this if you wish to use the output .H file in a file other than the one from which the .H file was generated.

PROTOTAG, The extracted header file contains prototypes with tag names, which should be moved to after the tag name declaration.

Description

When extracting function prototype declarations, the compiler has encountered a parameter type specifier that references a tag. Because the created prototype will use this tag, it should be moved after the tag declaration in the final compilation source. This message can only be generated when the compiler has been invoked with the option to extract function prototype declarations.

User Action

Be aware of this if you wish to use the output .H file.

PROTOTYPEDEF, The extracted header file contains prototypes with typedefs, which should be moved to after the typedef declaration.

Description

When extracting function prototype declarations, the compiler has encountered a parameter type specifier that is defined by a typedef. Because the created prototype will use this typedef, it should be moved after the typedef declaration in the final compilation source. This message can only be generated when the compiler has been invoked with the option to extract function prototype declarations.

User Action

Be aware of this if you wish to use the output .H file.

PROTOVLA, The extracted header file contains prototypes for functions which have formal parameters with variably modified type. All variable length bound specifiers have been replaced by a "*" signifying a variable length array of unspecified size.

Description

When extracting function prototype declarations, the compiler has encountered a function or functions which have a formal parameter with variably modified type. The compiler is unable to recreate the source that specified the number of array elements. Instead, the output prototype will use the "*" bounds specifier. Note that the output prototype is valid for the function.

User Action

Be aware that the compiler has made this minor change to the function declaration.

PSECTFIRST, "#pragma *psect_type*" directive must precede any declarations.

Description

The #pragma code_psect or #pragma linkage_psect directives must appear before any function or external data definitions.

User Action

Place the directive earlier in the source program.

PSECTTOOLONG, Psect name is too long (maximum is 31 characters). Pragma is ignored.

Description

A psect name specified in a #pragma code_psect, #pragma linkage_psect, or #pragma extern_model directive must be less than 32 characters in length. The compiler will ignore the directive.

User Action

Shorten the psect name.

PTRINTTOLONG, *context*"*expression*", a pointer to a 32-bit integer, is being cast to a pointer to a 64-bit integer. This may lead to unintended results.

Description

On many platforms long integers are the same size as integers, and casting a pointer to int to a pointer to long int is not a problem. On this platform long integers are 64-bits. This cast could indicate a potential porting problem.

User Action

Verify that this is the intended behavior.

PTRLONGTOINT, *context*"*expression*", a pointer to a 64-bit integer, is being cast to a pointer to a 32-bit integer. This may lead to unintended results.

Description

On many platforms long integers are the same size as integers, and casting a pointer to long int to a pointer to int is not a problem. On this platform long integers are 64-bits. This cast could indicate a potential porting problem.

User Action

Verify that this is the intended behavior.

PTRMISMATCH, *context*the referenced type of the pointer value "*expression*" is "*type*", which is not compatible with "*target type*".

Description

In a pointer assignment, the type pointed to by the source pointer is different than the type pointed to by the destination pointer.

User Action

Correct the assignment to use compatible types. This can be done by inserting a cast operand.

PTRMISMATCH1, *context*the referenced type of the pointer value "*expression*" is "*type*", which is not compatible with "*target type*" because they differ by signed/unsigned attribute.

Description

In a pointer assignment, the type pointed to by the source pointer is different than the type pointed to by the destination pointer. In this case the types differ because the signed/unsigned type attributes are different.

User Action

Correct the assignment to use compatible types. This can be done by inserting a cast operand.

QUALAFTCOMMA, Type qualifier(s) after a comma ignored.

Description

In Microsoft mode, the compiler used to accept a type qualifier after a comma used for separating declarators. Because Microsoft no longer accepts this type of declaration, VSI C will no longer accept it. The type qualifier is ignored.

User Action

Remove the type qualifier.

QUALFUNCRET, The return type of "*name*" is a qualified type. Type qualifiers have no meaning for function return values.

Description

A type qualifier has been used as part of the type of a function return value. The type qualifiers have no meaning for function return values.

User Action

Remove the type qualifier.

QUALISPTR, *context*"*expression*" has a pointer type, but occurs in a context that expects a struct or union.

Description

The left operand of the struct/union member operator (.) is a pointer type instead of a struct or union type.

User Action

Specify the correct struct or union type object as the left operand. In cases where the left operand is a pointer to a struct or union, it might be possible to use the struct/union pointer operator (->) instead of the member operator.

QUALNA, The *qualifier name* qualifier is not available on this platform and will be ignored.

Description

The specified qualifier is not supported on this platform.

User Action

Remove the qualifier from the command line.

QUALNOTUS, *context*the qualifier for "*name*" is not a struct or union.

Description

In certain modes, the compiler will allow the left operand of a struct/union member reference to be certain types other than a struct or union type. In these cases the compiler will issue a warning that this non-standard syntax is being accepted.

User Action

Modify the left operand to be a struct or union type.

QUESTCOMPARE, *context*the unsigned expression "*expr*" is being compared with a relational operator to a constant whose value is not greater than zero. This might not be what you intended.

Description

An ordered comparison between an unsigned value and a constant that is less than or equal to zero often indicates a programming error. Humans consider an unsigned value to be larger than any negative value. But in C a negative value is converted to an unsigned value before the comparison, so any negative value compares larger than most unsigned values. An ordered comparison of an unsigned value to zero suggests a programming error because the value can only be greater than or equal to zero. If the code is correct, the comparison could be more clearly coded by testing for equality with zero.

User Action

Cast (or otherwise rewrite) one of the operands of the compare to match the signedness of the other operand, or compare for equality with zero.

QUESTCOMPARE1, *context*the unsigned expression "*expr*" is being compared with an equality operator to a constant whose value is negative. This might not be what you intended.

Description

An unsigned value and a signed constant whose value is negative are being compared for equality. Logically, these value would never be equal. But in C the negative constant value is converted to an unsigned value before the comparison, and may well compare equal.

User Action

Cast (or otherwise rewrite) one of the operands of the compare to match the signedness of the other operand.

QUESTCOMPARE2, *context*the unsigned expression "*expr*" is being tested to see if it is greater than zero. This might not be what you intended.

Description

An ordered comparison between an unsigned value and a constant that is zero may indicate a programming error. Often C programmers do not realize that an expression has an unsigned type. If the code is correct, the comparison could be more clearly coded by testing for equality with zero.

User Action

Cast (or otherwise rewrite) one of the operands of the compare to match the signedness of the other operand, or compare for equality with zero.

READONLYEXT, readonly is a language extension.

Description

The readonly storage class modifier is a language extension of VSI C. Other C compilers might not successfully compile a program that uses the extension.

User Action

Be aware of this extension if you wish to port the code.

REDECLNOPARAM, *context*the declaration of the function "*name*" containing no parameter information replaces an earlier declaration of "*name*" at *location*.

Description

A function which was previously declared with a function prototype has been redeclared without parameter information. This is a violation of the C standard. The VSI C compiler will accept this for compatibility with older compilers.

User Action

Remove one of the declarations.

REDEF, This declaration contains a redefinition of "*name*". The previous declaration is at *location*.

Description

This declaration has tried to redefine an identifier that was defined earlier. This is not valid.

User Action

Remove one of the definitions.

REDEFSTRUCT, *context*the struct "*name*" is redefined.

Description

The struct tag declared in this declaration is already declared as a struct tag by another declaration.

User Action

Change the name of the struct tag.

REDEFTAG, *context*the tag "*name*" is redeclared.

Description

The tag declared in this declaration is already declared.

User Action

Change the name of the tag.

REDEFUNION, *context*the union "*name*" is redefined.

Description

The union tag declared in this declaration is already declared as a union tag by another declaration.

User Action

Change the name of the union tag.

REFBEFORETLS, *context*the reference to the variable "*var*" lexically precedes its use in a #pragma omp threadprivate directive. This is not allowed.

Description

An OpenMP threadprivate directive must lexically precede all references to any varible in its variable list. The compiler had detected a reference to a variable which appears in a subsequent threadprivate directive.

User Action

Move the threadprivate directive before the reference.

REGCONFLICT, Conflicting required uses of register(s): *text*

Description

The special linkage associated with a function has specified that one of the standard calling convention registers be used in a nonstandard way without also replacing its standard use with another register. An example would be a function that returns an int value using a special linkage that states R0 is not used, and does not specify another register to hold the return value.

User Action

Correct the #pragma linkage directive that specifies the special linkage.

REGNOSHARE, *context*noshare cannot be used with the register storage class. Modifier noshare is ignored.

Description

The storage class modifier noshare is meaningless for objects declared with register storage class. The compiler ignores the noshare.

User Action

Remove the noshare storage class modifier.

RELOCALIGNMENT, An initialization requiring relocation is not correctly aligned at Psect *text* + *number*

Description

On some platforms, initializing an object to an address requires that the object be aligned on a natural boundary.

User Action

Either remove the static initializer or align the object being initialized.

RESMISMATCH, The pointer size restored with pragma *pragma name* was saved using pragma *pragma name*. The pointer size restored will take effect.

Description

VSI C supports two forms of the pointer-size directives. One begins with #pragma pointer_size, the other with #pragma required_pointer_size. A program has mixed the required_pointer_size and the pointer_size form of the pointer-size directives in a way that is not recommended. This might indicate a programming error.

User Action

If a pointer size has been saved by one form of the pointer-size directive, it should be restored by the same form of the directive.

RESTRICTEXT, The __restrict type qualifier is a language extension.

Description

The use of the __restrict type qualifier might not be portable to other C compilers.

User Action

Be aware of this portability concern.

RESTRICTEXT1, Placement of the __restrict qualifier within the array-bound specifier of a formal parameter declaration is a language extension.

Description

The use of the restrict type qualifier within the array bound specifier of a formal parameter is a language extension supported by VSI C. Other C compilers might not successfully compile a program that uses this extension.

User Action

Be aware of this if you wish to port the program.

RESTRICTEXT2, The restrict type qualifier is a new feature in C99. Other C compilers might not successfully compile a program that uses this feature.

Description

The use of the restrict type qualifier might not be portable to other C compilers.

User Action

Be aware of this portability concern.

RESTRICTNOP, The restrict type qualifier can only be applied to a pointer type that points to an object or incomplete type. Qualifier is ignored.

Description

The restrict type qualifier has been used with an invalid type. Only pointers to object or incomplete types can have the restrict type qualifier. The compiler will ignore the type qualifier in all other cases.

User Action

Remove the type qualifier or change the type to one that accepts the qualifier.

RETLOCALADDR, This return statement returns the address of a local variable. The address returned cannot be used by the caller in any meaningful way.

Description

The storage for all local variables is undefined after a function has returned. Returning the address of a local variable will cause undefined behavior when the return value is used in the calling program.

User Action

Either change the variable to have static storage duration, use malloc to allocate the storage (and free it after its use), or change the interface to have the caller pass in the address at which data is to be stored.

RETRYCONV, Built-in function *retry-name* is not available on this platform. It has been converted to *nonretry-name* by ignoring the retry count and setting the retry status to 1.

Description

The version of this built-in function with retry capability is not available on the IA64 platform.

User Action

Use the non-retry version of this built-in function.

RETRYNOTAVAIL, Built-in function *name* with retry count is not available on this platform. The retry count is ignored.

Description

The retry capability of this built-in function is not available on the IA64 platform.

User Action

Remove retry count from built-in function call.

RETVALTOOBIG, The size of return value of "*name*" exceeds the maximum size of an object allowed on this platform which is *size* bytes.

Description

A function's return value is too large for this platform.

User Action

Reduce the size of the return value.

RIGHTSHIFTOVR, *context*the right shift count "*number*" is greater than or equal to the size of the unpromoted operand "*expression*".

Description

The compiler has detected a right shift count that is greater than or equal to the size of the operand to be shifted (before application of the integral promotions). This might not be what you intended, as the result contains none of the original bits of the operand. For an unsigned operand, the result is always 0. For a signed operand, the result is either 0 or -1, depending on whether or not the operand had a negative value. The same result would be achieved by shifting a signed operand one fewer bits.

User Action

Correct the shift count (or replace the expression by 0 if appropriate).

RTEXCEPT, *context*the floating-point constant named "*name*" will cause an exception at runtime.

Description

The IEEE trap mode of this program will cause an exception at runtime if this floating-point constant is used in an expression.

User Action

If you do not choose to cause a runtime exception, replace the named constant with a conventional floating point constant. The HUGE_VAL macros defined by <math.h> may be used in place of IEEE Infinities with any floating-point representation.

RTLMAPNOTFOUND, C RTL mapping information for RTL *name* not found. Could not access *image_name*.

Description

In most cases, the VSI C compiler will automatically map names of C standard library functions to their corresponding names in the VSI C RTL shareable image. In many cases, this is done simply by adding a "DECC$" prefix to the name. In order for this mapping to work, the compiler accesses an RTL mapping table. This message is issued if the compiler was unable to open the mapping table. In these cases, no name mapping will be performed. The most common cause of this message is specifying bad name in the /PREFIX=RTL="name" compiler qualifier.

User Action

Specify a valid RTL on the /PREFIX=RTL qualifier. If no qualifier was used, it might be necessary to reinstall the compiler and/or RTL. For more information consult the VSI C Run-time Library Manual for OpenVMS Systems.

RTLMISMATCH, VSI C RTL prefix table version mismatch: RTL table is V*major.minor*, compiler needs V*major.minor*.

Description

In most cases, the VSI C compiler will automatically map names of C standard library functions to their corresponding names in the VSI C RTL shareable image. In many cases, this is done simpl by adding a "DECC$" prefix to the name. In order for this mapping to work, the compiler accesses an RTL mapping table. The compiler also requires that the version of the RTL mapping table be compatible with the version of the compiler. In cases where the versions are incompatible, this message is generated. In these cases, no name mapping will be performed. The most common cause of this message is specifying an old RTL name in the /PREFIX=RTL="name" compiler qualifier.

User Action

Specify a new RTL on the /PREFIX=RTL qualifier. If no qualifier was used, it might be necessary to reinstall the compiler and/or RTL. For more information, consult the VSI C Run-time Library Manual for OpenVMS Systems.

SAMEASTYPEDEF, *context*the extern has the same name as a file-scope typedef. This is a language extension.

Description

The program has declared an extern inside a function whose name matches a file-scope typedef. This is not allowed by the C standard, but is accepted for compatibility with other C compilers.

User Action

Change the name of the variable or the typedef.

SCACALL, This function contains too many parameters for SCA to handle. Function parameter info will be truncated.

Description

The parameter information for this function contains more data than SCA can process. The compiler will truncate the parameter information. Be aware that the parameter information will be incomplete.

User Action

Simplify the parameter information.

SCAID2LONG, The identifier exceeds the SCA limit of *number* characters. In the SCA file the name will be truncated to "*truncated spelling*".

Description

The length of an identifier supported by SCA is less than the length of an identifier supported by the VSI C compiler. Because of this, the compiler will truncate an identifier name to fit the SCA limits.

User Action

Either reduce the identifier name, or be aware of this when using SCA.

SCALEFACTOR, The CDD description for *name* specifies a scale factor of *number*. The scale factor is being ignored.

Description

VSI C does not support scaled arithmetic.

User Action

Verify that all computations involving this item are correctly scaled.

SCAOVFLO, Compiler Internal Error: SCA event buffer overflowed. Please submit a problem report.

Description

When building SCA information, the compiler overflowed its internal buffer. This should not happen.

User Action

Please submit a problem report detailing the failure.

SEQUENCEEXT, *context*allowing a comma operator is a language extension.

Description

In this context the C standard does not allow the comma (sequence) operator. VSI C allows this syntax for compatibility with some other C compilers. Be aware that this syntax may not be accepted by other C compilers.

User Action

If the intent is to use an expression that is not necessarily a constant expression, then enclose it in parentheses. But if the intent of the declaration is to use a constant expression, then the comma operator cannot be used.

SESEMULTIEXITS, parallel directive scope *text* has multiple exits

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

SESEMULTIPREDS, parallel directive scope *text* has multiple entry paths

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

SESEVFLOW, parallel directive scope *text* is crossed by a VBRANCH

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

SHARECONST, In this declaration, noshare has been ignored due to the presence of const or readonly.

Description

A variable declared with the readonly storage class modifier, or the const type modifier cannot also have the noshare storage class modifier. The compiler will ignore the noshare storage class modifier.

User Action

Remove either the noshare storage class modifier or the const or readonly modifiers.

SHIFTCOUNT, *context*the shift count "*number*" is negative or is greater than or equal to the promoted size of the operand "*expression*".

Description

The compiler has detected a shift count that is negative or is greater than or equal to the promoted size of the operand to be shifted. This behavior is undefined.

User Action

Correct the shift count.

SHORTCIRCUIT, *context*potential side effects from the evaluation of "*operand*" will not take place. This is because the first operand of a logical operator is a constant whose value requires that this expression must not be evaluated.

Description

The C language requires that if the first operand of a logical || or && operator determines the result of the expression, the second operand must not be evaluated. This behavior is different from other operators. The compiler has noticed that the second operand will generate code that may produce side effects that the programmer expects to take place. This message is to inform the user that the code generated for the second operand will not be executed.

User Action

Replace the logical expression with its first operand.

SHOWMAPLINKAGE, The linkage has been mapped to: #pragma linkage_ia64 *name* = (*stuff*).

Description

The pragma linkage directive contains architecture-specific information. The Alpha register conventions are different from the IA64 register conventions. The compiler will try to map the Alpha registers to the corresponding registers on IA-64. This message details the mapping.

User Action

Replace the linkage directive with the linkage_ia64 directive that appears in the message.

SIGNEDKNOWN, *context*HP C recognizes the standard keyword "signed". This differs from the VAX C behavior.

Description

VAX C does not recognize the "signed" keyword. VSI C will allow this, even in vaxc mode.

User Action

Be aware of this difference if you plan to compile the source with VAX C.

SIGNEDMEMBER, *context*HP C recognizes the standard C keyword "signed" in member declarations. The VAX C compiler does not and would treat the member as unsigned.

Description

VAX C does not recognize the "signed" keyword in a member declaration. VAX C will treat the member as an unsigned type. VSI C will recognize the keyword and declare the member as a signed type.

User Action

Be aware of this difference if you plan to compile the source with VAX C.

SIMPLEMESSAGE, *user text*

Description

The compiler has encountered a #pragma message () directive. It will output the message in the quoted string.

User Action

Remove the pragma message.

SIZEBIT, *context*"*expression*" is a bitfield, and so has no size.

Description

A bitfield expression cannot be used as the argument to the sizeof operator or the __builtin_alignof builtin.

User Action

Pass an expression with a valid type to the operator or builtin.

SIZEINCOMP, *context*"*expression*" is of an incomplete type, and so has no size.

Description

An expression that has incomplete type has no size and therefore cannot be used as the argument to the sizeof operator.

User Action

Pass an expression with a valid type to the sizeof operator.

SIZEINCOMPTYP, *context*"*type*" is an incomplete type, and so has no size.

Description

A incomplete type has no size and therefore cannot be used as the argument to the sizeof operator.

User Action

Pass a valid type to the sizeof operator.

SIZFUNVOIDTYP, *context*"*type*" has function or void type and may not appear in this context. The compiler will treat the type as if it were char.

Description

A function or void type cannot be used as the argument of the sizeof operator or the __builtin_alignof builtin. For compatibility with some other compilers, an output file is still created. The result produced will be the same as if a char type was passed. This may or may not be compatible with other compilers that accept this syntax.

User Action

Pass a valid type to the operator or builtin.

STACKPOP, This "restore" has underflowed the pragma *stack name* stack. No corresponding "save" was found.

Description

One of the pointer-size stacks, managed by the #pragma pointer_size, #pragma require_pointer_size, #pragma required_vptr_size, and #pragma environment directives, contains more restores than saves. This could signify a coding or logic error in the program.

User Action

Make sure each restore has a corresponding save.

STATICIFLOAT, *context*conversion of a link-time address constant to a floating type is required. This is not allowed.

Description

The initialization of an object with static extent requires a value that is a link-time constant expresion. Link-time constant expressions cannot involve values of floating types (other than floating constants that are the immediate operands of casts).

User Action

Remove the floating point types from the initialization.

STATICVLA, *context*the static object "*name*" cannot be a variable length array.

Description

Only ordinary identifiers with block scope and without storage class extern or static, or ordinary identifiers with function prototype scope can be declared as variable-length arrays.

User Action

Correct the declaration.

STATINITWARN, *context*the linker will be unable to perform this static initialization if the initializer is defined in a sharable image.

Description

A static initialization will require that a link-time constant be truncated. If the constant is resolved in a sharable image, the linker will issue a diagnostic and be unable to perform the initialization. This message is output on OpenVMS systems only.

User Action

Rewrite the static initialization so that the link-time constant will not be truncated.

STDARG, *context*stdarg.h macros might be required if the address of the parameter *name* is used to index through a parameter list.

Description

Some older C programs will traverse a function's parameter list by taking the address of one of the parameters and then adjusting it to get to subsequent parameters. In most cases, this technique will not produce the desired results on Alpha. This message is specific to UNIX, and is only output if -varargs option is specified.

User Action

If the address is used to walk the parameter list, recode the function to use the standard stdarg.h macros.

STKALLEXC, Allocations to stack exceeded maximum stack size

Description

A routine uses more stack space than is available on this platform. This is most often caused by declaring too many large automatic variables.

User Action

Reduce the size required by the automatic variables.

STOALNERR, Psect *text* alignment is insufficient for allocation of *text*

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

STONOTFIRST, The placement of a storage-class specifier other than at the beginning of the declaration specifiers in a declaration is an obsolescent feature.

Description

The standard states that this style of declaration is obsolescent.

User Action

Place the storage-class specifier first in the declaration.

STORCLSDCL, *context*a storage class without a declarator is meaningless.

Description

This message is generated when the compiler encounters certain declarations that contain a storage class but no declarator. For example: extern struct S { int a;};

User Action

Either remove the storage class or add a declarator to the declaration.

STOREBIF, Built-in function *store-bif* is not available on this platform. It may be converted to *swap-bif* if the source and dest parameters are identical.

Description

The STORE version of this built-in function is not available on the IA64 platform.

User Action

Use the SWAP version of this built-in function.

STOREQEXC, Allocations to *text* section exceeded growth bounds

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

STORISSTAT, This redeclaration of the static initialized variable "*name*" will have static storage class that differs from the VAX C behavior. The previous declaration is at *location*.

Description

In VAX C mode, if a variable is first declared static and then declared extern, the variable will be given extern storage class. This matches the VAX C behavior. If, however, the static variable is initialized, the storage class will remain static.

User Action

Be aware of this difference.

STORMODDCL, *context*a storage class modifier without a declarator is meaningless.

Description

This message is generated when the compiler encounters certain declarations that contain a storage class modifier but no declarator. For example: readonly struct S { int a;};

User Action

Either remove the storage class modifier or add a declarator to the declaration.

STRCTPADDING, An additional *number* bytes of padding have been implicitly inserted prior to this member for proper alignment of this member.

Description

The compiler has added pad bytes before a member so that it will be accessed efficiently. This might not have been what you intended.

User Action

Consider rearranging the order of member declarations.

STRINGCONST, Ill-formed string constant.

Description

An invalid string constant was encountered.

User Action

Correct the string constant.

STRUCTBRACE, *context* a required set of braces is missing.

Description

The initializer for this struct was not enclosed in braces. While some compilers allow this, standard C requires braces around the initializer.

User Action

Enclose the initializer in braces.

STRUCTLIMITSUP, *context*HP C provides only limited support for struct/union types larger than *n* bytes.

Description

This struct/union type is larger than can be represented by size_t. While VSI C will allow a type declared to be this size, uses of the type are not fully supported and may cause unpredictable behavior.

User Action

Reduce the size of the type.

STRUCTOVERFLOW, Integer overflow occurred when computing the size of a struct or union type.

Description

An struct or union type is larger than allowed on this platform. Note that as the compiler computes the size of the type in bits, the limit on the size of struct/union types is eight times smaller than the size of other types.

User Action

Reduce the size of the struct/union type.

SUBINVALIDCHR, Parameter substitution produced an invalid character constant.

Description

In certain modes, the compiler will replace identifiers found within a character constant if they match a macro argument name. This form of "old-style stringization" is provided for compatibility with older C compilers. This message is output if this replacement forms an invalid character constant.

User Action

Modify the macro argument so that a valid character constant is formed.

SUBINVALIDSTR, Parameter substitution produced an invalid string literal.

Description

In certain modes, the compiler will replace identifiers found within a string literal if they match a macro argument name. This form of "old-style stringization" is provided for compatibility with older C compilers. This message is output if this replacement forms an invalid string literal.

User Action

Modify the macro argument so that a valid string is formed. VSI also recommends that the macro body be rewritten to use the standard C stringize operator (#).

SUBSCRBOUNDS, *context*an array is being accessed outside the bounds specified for the array type.

Description

The compiler has detected an array access that is outside the bounds of the array. The array access might cause unpredictable behavior. Note that in C, an array is declared using the number of elements, but the first element has subscript 0. It is a common coding error to attempt to access the last element of an array of "n" elements using a subscript of "n" instead of "n - 1". However, there are two common practices that intentionally employ out-of-bounds subscripts to useful/correct effects that are not reported by this message, but have separate optional messages. First, taking the address of an array element that is exactly one beyond the last element of an array is completely valid in standard C as long as the address is not used to access memory. The optional subscrbounds2 message can be enabled to report taking the address of the array element exactly one beyond the last element. Second, it is a somewhat common practice to declare the last member of a struct as an array with one element, and then allocate such structs at runtime with different sizes, recording the actual size in an earlier member of the struct. The optional subscrbounds1 message can be enabled to report subscripts greater than zero applied to arrays declared with only one element.

User Action

Specify an array subscript that is within the bounds of the array type.

SUBSCRBOUNDS1, *context*an array type declared with one element is being accessed beyond the end of the array.

Description

An array declared with one element is being accessed beyond the end of the array. The array access can cause unpredictable behavior. Note that in C, an array is declared using the number of elements, but the first element has subscript 0. It is a common coding error to attempt to access the last element of an array of "n" elements using a subscript of "n" instead of "n - 1".

User Action

Specify an array subscript that is within the bounds of the array type.

SUBSCRBOUNDS2, *context*accessing the address of an array element that is exactly one beyond the end of the array might not be what you intended.

Description

Accessing the address of an array element that is exactly one beyond the end of the array might be a coding error (e.g. if the address is then used to access memory), or it might be fully correct (e.g. to compute a pointer value to be used as the upper bound on a loop).

User Action

Specify an array addressthat is within the bounds of the array type.

SWAPBIF, Built-in function *store-bif* is not available on this platform. The compiler was able to convert it to *swap-bif* because the source and dest parameters are identical.

Description

The STORE version of this built-in function is not available on the IA64 platform.

User Action

Use the SWAP version of this built-in function.

SWITCHLONG, The signed or unsigned long expression "*expression*" is used in a switch statement.

Description

A switch value has an integer type of signed or unsigned long int. While this is perfectly portable under the C standard, the original K&R definition of C required that the expression have type int. VSI C accepts this usage in all modes, but there may be older C compilers that require type int in this context.

User Action

Be aware that older, non-standard compilers might not accept this construct, or force the result to type int.

SYSREGUSED, System register specified as external register.

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

SYSTEM, *text*

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

TAGDIFFER, *context*the tag "*name*" differs from the tag "*name*" used in an earlier declaration of this function.

Description

This function declaration contains a parameter that is a pointer to a struct or union type whose tag differs from the tag of the struct or union type referenced by a pointer type in an earlier declaration of this function. In most modes of the compiler, this will cause the function declarations to be incompatible.

User Action

Multiple declarations of a function should use exactly the same parameter types.

TAGORBRACE, Missing tag or "{".

Description

The enum, struct, and union keywords must be followed by either an open brace or a tag.

User Action

Correct the program syntax.

TENTREDEF, This *definition or tentative definition* of "*name*" is redefining the *definition or tentative definition* on *location*. This is not allowed in C++. *compiler__declare_tent_redef1*.

Description

C will allow a tentative definition to be redefined later in the compilation unit. C++ does not have the concept of a tentative definition. Therefore this redefinition is invalid in C++.

User Action

Either remove the previous tentative definition, or modify it to match the later redefinition.

TEXTARRAY, The CDD description for *name* specifies that it is text 1; It has been translated into an array of char.

Description

When the CDD type "TEXT" is of length one, VSI C will normally convert this to type "char" in order to be compatible with VAX C. If however, the nullterminate or text1toarray keywords are specified in a VSI C dictionary directive, the CDD type TEXT will be converted to type "array of char".

User Action

No action is necessary as long as the type "array of char" is the desired datatype.

TEXTARRAYN, The CDD description for *name* specified that it is text 1; It has been translated into an array of char because null_terminate was used.

Description

The CDD type "TEXT" is of length 1 and has been converted to the C type "array of char" of size 2 because the null_terminate keyword was specified on the dictionary directive.

User Action

No action is necessary as long as the type "array of char" of size two is the desired C datatype.

TEXTCHAR, The CDD description for *name* specified that it is text 1; It has been translated into a type char.

Description

When the CDD type "TEXT" is of length one, VSI C will normally convert this to type "char" in order to be compatible with VAX C. However, when the length of the CDD type "TEXT" is greater than one, the C type "array of char" is generated. This means that CDD type "TEXT" will result in different C datatypes depending upon the length of the TEXT stored in the dictionary.

User Action

No action is necessary as long as the type "char" is the desired datatype.

TEXTMODULE, The text library module form of #include is an extension.

Description

On OpenVMS systems, an #include directive whose file specifier is not enclosed in either quotation marks or angle brackets denotes an include from a text library. This is an extension of VSI C. This directive will not work as expected on non-VMS systems.

User Action

Be aware of this if you wish to port the program.

THREADFUNC, *context*the __declspec(thread) storage class modifier cannot be used with a function type. Modifier is ignored.

Description

Only objects can be declared with thread-local storage. The storage class modifier is ignored when applied to an identifier with function type.

User Action

Either remove the storage class modifier or change the type to one that is valid for __declspec(thread).

THREADNYI, *context*the __declspec(thread) storage class modifier is not implemented on this platform. It will be ignored except to verify correct compile-time usage.

Description

Thread-local storage is only supported on UNIX platforms. The compiler will ignore the storage class modifier except to verify that it is correctly used.

User Action

Remove the __declspec(thread) storage class modifier or compile the program on a platform that supports the modifier.

THREADSTO1, *context*the __declspec(thread) storage class modifier requires a storage class of extern, static, or none. Modifier is ignored.

Description

Declaring a file-scope object with thread-local storage requires that the object have a storage class of extern, static or none. In other cases, the compiler will ignore the __declspec(thread) storage class modifier.

User Action

Either remove the storage class modifier or change the storage class to one that is valid for __declspec(thread).

THREADSTO2, *context*the __declspec(thread) storage class modifier requires a storage class of extern, or static. Modifier is ignored.

Description

Declaring a local object with thread-local storage requires that the object have a storage class of extern, or static. In other cases, the compiler will ignore the __declspec(thread) storage class modifier.

User Action

Either remove the storage class modifier or change the storage class to one that is valid for __declspec(thread).

TLSANDSTATIC, *context*the storage class modifier __declspec(thread) cannot be used with the -static option. The storage class modifier is ignored.

Description

Thread-local storage cannot be declared in compilations that are performed with the -static option. The compiler will ignore the __declspec(thread) storage class modifier.

User Action

Either remove the __declspec(thread) storage class modifier or do not compile with the -static option.

TOOFEWACTUALS, Too few actual parameters in the invocation of the macro "*name*".

Description

A macro invocation supplied fewer actual arguments than the macro expects. The macro arguments not specified in the call will be given a null value.

User Action

Supply all arguments in the macro invocation.

TOOFEWARGS, *context*"*function expression*" expects *correct number* arguments, but *actual number* are supplied.

Description

A function has been invoked with fewer arguments than it expects. In some modes this is a warning message, and the compiler will compile the program. In this case, the function being called might not produce the expected results.

User Action

Make sure the number of arguments passed to a function match those specified in the function declaration.

TOOFEWARGSO, *context*"*function expression*", which was declared with an old-style function definition, expects *correct number* arguments, but *actual number* are supplied.

Description

A function that was declared with an old-style function definition has been invoked with fewer arguments than it expects. While this is valid C, it might not have been what you intended.

User Action

Make sure the number of arguments passed to a function match those specified in the function declaration. If the function is to be called with a variable number of arguments, it should use the facilities of for old-style definitions. VSI generally recommends that old-style function definitions be replaced by prototype-format definitions, in which case variable argument lists are specified using the ... notation and the definition uses the facilities of .

TOOLONG, *context*, "*expression*" is too long by *count* character(s).

Description

A string initializer for a char array contains more characters than the array can hold. This is not valid.

User Action

Reduce the number of characters to be less than or equal to the number of elements in the char array.

TOOMANY, *context*, there are *actual number* elements, which is *extra number* too many. The extra initializers will be ignored.

Description

An initializer list contains more initializers than there are objects to be initialized. This is not valid.

User Action

Reduce the number of initializers to be less than or equal to the number of objects being initialized.

TOOMANYACTLS, Too many actual parameters in the invocation of the macro "*name*".

Description

A macro invocation supplied more actual arguments than the macro expects. The additional arguments will be ignored.

User Action

Remove the extra arguments from the macro invocation.

TOOMANYARGS, *context*"*function expression*" expects *correct number* arguments, but *actual number* are supplied.

Description

A function has been invoked with more arguments than it expects. In some modes this is a warning message, and the compiler will compile the program.

User Action

Make sure the number of arguments passed to a function match those specified in the function declaration.

TOOMANYARGSO, *context"function expression"*, which was declared with an old-style function definition, expects *correct number* arguments, but *actual number* are supplied.

Description

A function that was declared with an old-style function definition has been invoked with more arguments than it expects. While this is valid C, it might not have been what you intended.

User Action

Make sure the number of arguments passed to a function match those specified in the function declaration. If the function is to be called with a variable number of arguments, it should use the facilities of for old-style definitions. VSI generally recommends that old-style function definitions be replaced by prototype-format definitions, in which case variable argument lists are specified using the ... notation and the definition uses the facilities of .

TOOMANYERR, More than *number* errors were encountered in the course of compilation.

Description

After emitting a certain number of errors, the compiler will stop the compilation and issue this message. The number of errors output before the compilation stops can be changed using the /ERROR_LIMIT qualifier on OpenVMS systems, or the -error_limit option on UNIX systems.

User Action

Either reduce the number of errors generated by the program or give a larger value for the error limit.

TOOMANYGATES, only 64 gates maybe be used within a parallel region

Description

For each parallel region there is a limit of 64 different gates that can be specified in a #pragma enter gate/#pragma exit gate pair.

User Action

Reduce the number of gates

TOOMANYTOKENS, Too many tokens in macro expansion.

Description

An argument to the #line preprocessing directive contained a macro whose expansion generated more tokens than the #line directive expects.

User Action

Either modify the macro definition or change the arguments to the #line directive.

TOOMANYTXTLIB, Too many text libraries. Library *library name* and subsequent will not be searched.

Description

The compiler has tried to open more text libraries than it can support in its internal data structures. The specified library, and all subsequent libraries will not be opened.

User Action

Reduce the number of text libraries the compilation requires.

TOOMNYREL, Object file section *text* has *number* relocations; maximum allowed is *number*

Description

This message is emitted by the code generator. It should never be output when compiling a C program.

User Action

Please submit a problem report if you encounter this message when compiling a C program.

TRAILCOMMA, Trailing comma found in enumerator list.

Description

Accepting an enumerator list that contains a trailing comma is an extension of VSI C provided for compatibility with other C compilers. An enumerator list with a trailing comma is not valid in C89, nor in C++. The C99 standard does permit this syntax.

User Action

Remove the trailing comma.

TRUNCFLTASN, *context*"*expression*" has more precision than "*target type*". Assignment might result in loss of precision and/or range.

Description

The destination of a floating-point assignment has less range and/or precision than the expression being assigned to the destination. Because of this, the assignment might cause a loss of range and/or precision.

User Action

Verify that no unexpected data can be lost by the assignment. If not, cast the expression to the type of the destination.

TRUNCFLTINT, *context*"*expression*" is a floating-point type being assigned to an integer type. The assignment might result in data loss.

Description

A floating-point expression is being assigned to an integer type. This assignment might cause a loss of range and/or precision.

User Action

Verify that no unexpected data can be lost by the assignment. If not, cast the expression to the type of the destination.

TRUNCINTASN, *context*"*expression*" has a larger data size than "*target type*". Assignment might result in data loss.

Description

The destination of an integer or pointer assignment is smaller than the expression being assigned to the destination. Because of this, the assignment might cause data to be lost.

User Action

Verify that no unexpected data can be lost by the assignment.

TRUNCINTCAST, *context*"*expression*" has a larger data size than "*target type*". Cast might result in data loss.

Description

An integer or pointer expression is being cast to a size that is smaller than the expression. Because of this, the cast might cause data to be lost.

User Action

Verify that no unexpected data can be lost by the cast.

TRUNCLONGCAST, *context*"*expression*", a 64-bit integer, is being cast to a 32-bit integer. The cast might result in data loss.

Description

On many platforms long integers are the same size as integers. On this platform long integers are 64-bits. This cast could indicate a potential porting problem.

User Action

Verify that no unexpected data can be lost by the cast.

TRUNCLONGINT, *context*"*expression*", a 64-bit integer, is being assigned to a 32-bit integer. Assignment might result in data loss.

Description

On many platforms long integers are the same size as integers. On this platform long integers are 64-bits. This assignment could indicate a potential porting problem.

User Action

Verify that no unexpected data can be lost by the assignment.

TUNEOVERRIDE, tune setting *text* overridden by arch setting *text*, tune forced to *text*

Description

The program has specified a tune architecture that is older than the arch setting. The arch setting is the oldest architecture that the code should ever run on. Asking the compiler to tune for an even older architecture is not reasonable. The compiler will use the arch setting for the tune option as well

User Action

Specify a tune architecture that is at least as new as the arch architecture.

TYPEALIGN, *context*_align cannot be used with the typedef storage class. Modifier _align is ignored.

Description

The storage class modifier _align is meaningless for typedefs. The compiler ignores the _align.

User Action

Remove the _align storage class modifier.

TYPECONFLICT, *context*"*typespec1*" cannot be combined with "*typespec2*".

Description

Two type keywords used in the same type specifier cannot be combined. In some modes, the compiler will use the most recent keyword as the type specifier.

User Action

Correct the type specifier.

TYPEDEFFUNC, In this function definition, "*name*" acquires its type from a typedef.

Description

A function definition acquires its type from a typedef. This is not allowed.

User Action

Correct the function definition.

TYPEDEFINIT, The declaration of the typedef "*name*" contains an initializer. The initializer is ignored.

Description

A typedef declaration must not contain an initializer.

User Action

Remove the initializer from the declaration.

TYPEDEFNA, Accepting an old-style parameter name that matches a typedef is a language extension.

Description

The VSI C compiler will allow old-style parameters to have the same name as a typedef. Many other compilers will not allow this.

User Action

Recode the function definition to use the standard C prototype syntax.

TYPEDEFNOTDEF, In this declaration, "*name*" appears to be used as if it named a type, but there is no declared type of that name visible.

Description

The compiler has encountered what appears to be a typedef declaration that provides a new name for an existing type, but the identifier used to specify the existing type is not the name of a type that is visible.

User Action

Declare the identifier for the first type, or correct its spelling.

TYPEEXPR, *context*"*name*" is declared as *a label, tag, or typedef*, and so cannot occur as an expression.

Description

An identifier declared as a typedef has been used in an expression when an object or function was required.

User Action

Correct the expression.

TYPEOFEXT, The use of __typeof__ is a language extension.

Description

Support for __typeof__ is a language extension provided for compatibilty with some other C compilers. Although some other C compilers will accept this syntax, many compilers will reject it.

User Action

Be aware of this difference if you plan to port this source to another compiler.

TYPQUALNOT, A type qualifier is not allowed in this context.

Description

In Microsoft mode, the compiler used to accept a type qualifier after a comma used to separate declarators. This was refered to as a local type qualifier. This message is output when a local type qualifier is applied to a declarator that can not be qualified.

User Action

Remove the local type qualifier because this is no longer accepted.

TYPQUALNOT2, Use of the keyword "static" or a type qualifier within the outermost array-bounds specifier of a formal parameter declaration is a new feature in the C99 standard.

Description

The C99 construct may not be available in other compilers you use to build your application, in which case they will likely report it as a syntax error.

User Action

You may want to conditionalize your code with the preprocessor so that you can take advantage of the feature on platforms that support it, without getting syntax errors from older compilers or language modes that do not support it.

TYPQUALNOT3, Use of the keyword "static" or a type qualifier in an array-bounds specifier is invalid in this compilation mode. Keyword/qualifier ignored.

Description

Use of the keyword "static" or a type qualifier within the outermost array bound specifier of a formal parameter is a new feature in the C99 standard and is not supported in this language mode.

User Action

Either compile in a mode that supports C99 features, or remove the construct from your code.

TYPQUALNOT4, Use of this type qualifier in an array-bounds specifier is invalid. Qualifier ignored.

Description

Use of this type qualifier is not a permitted in the array bound specifier of a formal parameter.

User Action

Remove the keyword.

UABORT, Compilation terminated by user.

Description

This message is often output when the compilation was aborted by the user by hitting Control C.

User Action

Do not abort the compilation.

UCNICONVOPN, The call iconv_open(*CODESET*, "UCS-4") failed because: *STRERROR*. UCNs will not be mapped to the native character set.

Description

To translate Universal Character Name escape sequences to the codeset of the current locale, the compiler needs to call the iconv_open library routine with the specified parameters. This call failed, for the reason shown. Thus no UCN escape sequences in this program can be translated.

User Action

Make sure your system has the specified codeset converter installed, or set your locale to use a codeset for which a converter from UCS-4 is available. Alternatively, change your code to avoid the use of UCNs, e.g. using hexadecimal escape sequences.

UCNNOMAP, A UCN escape sequence was recognized, but there was no translation for it into the current codeset. The escape sequence will be used verbatim.

Description

A Universal Character Name (UCN) escape sequence was recognized, but there was no translation for it into the current codeset using the iconv library routine. The complete escape sequence itself, including the backslash, will be used in the object module.

User Action

Make sure your locale is set at compile-time to use a codeset for which a converter from UCS-4 is available, and which supports all of the characters that are expressed as UCNs in your program. Alternatively, change your code to avoid the use of UCNs, e.g. using hexadecimal escape sequences.

UCNUNSUPP, An apparent UCN escape sequence was encountered, but UCNs are not supported in this language mode. The backslash will be ignored.

Description

Universal Character Name (UCN) escape sequences were added to C in the C99 standard. The language mode of the current compilation does not process UCNs, so they will be treated as unrecognized escape sequences, which ignore the backslash.

User Action

Compile in a mode that processes UCNs (C99, or the default "relaxed" mode), or remove the backslash. Relying on apparent escape sequences to be unrecognized is not good practice.

UCNUSED, A UCN escape sequence was encountered.

Description

Universal Character Name (UCN) escape sequences were added to C in the C99 standard, and are processed in this language mode. C compilers and dialects that do not specifically process UCNs will treat them as unrecognized escape sequences, and may silently ignore the backslash.

User Action

Be aware of this if you wish to port the program.

UNALIGNEDFUNC, Ignoring __unaligned type qualifier in declaration of *name*.

Description

The __unaligned type qualifier has no meaning for function types. It is being ignored.

User Action

Remove the type qualifier.

UNALIGNEXT, The __unaligned type qualifier is a language extension.

Description

The use of the __unaligned type qualifier might not be portable to other C compilers.

User Action

Be aware of this portability concern.

UNAVAILPRAGMA, The pragma "*pragma name*" is not available on this platform.

Description

The compiler has encountered a pragma that is not currently supported on this platform. The compiler will ignore the pragma.

User Action

Compile the program on a platform that does support the pragma. Otherwise, understand that this pragma will have no effect.

UNAVOLACC, volatile access appears unaligned, but must be aligned at run-time to ensure atomicity and byte granularity

Description

The compiler has detected an unaligned access to a volatile variable. In order to meet atomicity and granularity requirements of volatile, the access will be done using an aligned instruction. This may cause an alignment fault at runtime if the access is unaligned.

User Action

Make sure volatile objects are aligned on a natural boundary.

UNCALLED, routine *text* can never be called

Description

The compiler has detected a static function that is never referenced.

User Action

Remove the unused function.

UNDECLARED, *context*"*name*" is not declared.

Description

An identifier used in an expression has not been declared. The only time an identifier can be used and not previously declared is when the identifier specifies the function name in a function call.

User Action

Either declare the identifier or remove its use.

UNDECLFUN, There is no function declaration for the identifier "*name*" at the point of this #pragma *pragma type attributes*.

Description

An identifier specified in a #pragma assert/hint func_attrs directive must refer to a function declaration at the point of the pragma.

User Action

Either remove the identifer from the pragma, correct its spelling, or reorder the source to ensure that a declaration of the identifier as a function is visible at the point of the pragma. Identifier must be a function declaration; no other kind of declaration (i.e. typedef, var, etc.) is allowed for func_attrs.

UNDECLVAR, There is no global declaration visible for the variable "*name*" at the point of this #pragma assert global_status_variable.

Description

An identifier specified in a #pragma assert directive must refer to a global variable declaration visible at the point of the pragma. The identifier will be ignored.

User Action

Either remove the identifer from the pragma, correct its spelling, or reorder the source to ensure that a declaration of the identifier as a global_variable is visible at the point of the pragma.

UNDEFENUM, *context*the enum "*name*" is not defined.

Description

The enum tag used to declare an enum variable is not defined at this point in the compilation.

User Action

Define the enum tag before using it.

UNDEFESCAP, An undefined escape sequence was encountered; the backslash is being ignored.

Description

The character or characters following a backslash do not form a valid escape sequence. The compiler will ignore the backslash.

User Action

Correct the escape sequence.

UNDEFINEDTYPE, The compiler was expecting a "*token*", but one was not found. This condition could have occured because "*id*" is used in what might be a type cast, but there is no declared type of that name visible.

Description

The compiler has discoverd a syntax error. This error may have been caused because a cast operator used an unknown type.

User Action

Correct the syntax error.

UNDEFVARFETCH, *context*the expression "*expr*" modifies "*var*", and fetches its value in a computation that is not used to produce the modified value without an intervening sequence point. This behavior is undefined.

Description

The compiler has detected a case where the same variable has been modified and fetched in a computation that does not later modify that same variable. Because the order of the variable fetch and store is not defined, this expression might produce different results on different platforms.

User Action

Rewrite the expression so that if a variable is stored to, it is fetched only to determine the value to be stored.

UNDEFVARMOD, *context*the expression "*expr*" modifies the variable "*var*" more than once without an intervening sequence point. This behavior is undefined.

Description

The compiler has detected a case where the same variable has been modified more than once in an expression without a sequence point between the modifications. Because what modification will occur last is not defined, this expression might produce different results on different platforms.

User Action

Rewrite the expression so that each variable is modified only once.

UNDERFLOW, *context*underflow occurs in evaluating the expression "*expression*".

Description

A floating-point underflow occurred while evaluating a constant expression. The value of the expression is undefined.

User Action

Correct the floating-point constant expression.

UNINIT1, The scalar variable "*var*"*declared in* is fetched but not initialized*inlineinfo*. And there may be other such fetches of this variable that have not been reported in this compilation.

Description

A variable's value has been used without being set. This might not have been what you intended. The algorithms that detect this situation only report it once for a given variable, and not necessarily at the first use of the uninitialized value.

User Action

Provide the variable with a value before the variable is used. If you only provide a value for the use reported here, you may find that when you recompile your program another uninitialized use is detected. It is best to initialize variables as close as possible to the point of declaration.

UNINIT2, Part or all of the non-scalar variable "*var*"*declared in* is fetched but not initialized*inlineinfo*. And there may be other such fetches of this variable that have not been reported in this compilation.

Description

A non-scalar variable has had its value used and some or all of the variable has not been given a value. This might not have been what you intended. The algorithms that detect this situation only report it once for a given variable, and not necessarily at the first use of the uninitialized value.

User Action

Provide the variable with a value before the variable is used. If you only provide a value for the use reported here, you may find that when you recompile your program another uninitialized use is detected. It is best to initialize variables as close as possible to the point of declaration.

UNINIT3, Variable "*var*"*declared in* is fetched but not initialized*inlineinfo*. And there may be other such fetches of this field that have not been reported in this compilation.

Description

The specified member of a struct variable has been used without being set. This might not have been what you intended. The algorithms that detect this situation only report it once for a given field, and not necessarily at the first use of the uninitialized value.

User Action

Provide the struct member with a value before the variable is used. If you only provide a value for the use reported here, you may find that when you recompile your program another uninitialized use is detected. It is best to initialize variables as close as possible to the point of declaration.

UNINIT4, Byte offsets *start* to *end* of "*var*"*declared in* are fetched but not initialized*inlineinfo*. And there may be other such fetches of this field that have not been reported in this compilation.

Description

The specified byte offsets of a variable have been used without being set. This might not have been what you intended. The algorithms that detect this situation only report it once for a given field, and not necessarily at the first use of the uninitialized value.

User Action

Provide the full variable with values before the variable is used. If you only provide a value for the use reported here, you may find that when you recompile your program another uninitialized use is detected. It is best to initialize variables as close as possible to the point of declaration.

UNINIT5, *fragment uninit5ainlineinfo*. Also the variable itself is not initialized. And there may be other fetches of this variable that have not been reported in this compilation.

Description

The specified storage location has been used without being set. This might not have been what you intended. In addition, as this fetch is outside the storage allocated to the variable, the behavior is undefined.

User Action

First verify that the fetch is correct(code that uses the address of a declared object to access memory outside the address range allocated to that object is not likely to be reliable). Then initialize the storage being fetched and, if necessary, the variable noted in the message. If you only provide a value for the use reported here, you may find that when you recompile your program another uninitialized use is detected, since the algorithms that detect this situation only report it once for a given variable, and not necessarily at the first use of the uninitialized value. It is best to initialize variables as close as possible to the point of declaration.

UNIONBRACE, *context* a required set of braces is missing.

Description

The initializer for this union was not enclosed in braces. While some compilers allow this, standard C requires braces around the initializer.

User Action

Enclose the initializer in braces.

UNKEXTMOD, Unknown extern model. Pragma is ignored.

Description

The compiler was unable to parse a #pragma extern_model directive. The extern_model must be an identifier that specifies one of the valid extern models. The directive will be ignored.

User Action

Correct the directive.

UNKINTRIN, The function "*routine name*" is not a known intrinsic function and cannot be used with #pragma intrinsic. Pragma not applied to this function.

Description

A function identifier specified in a #pragma intrinsic directive is not a valid intrinsic function on this platform. The pragma will not be applied to this identifier, leaving it to be treated as an ordinary function.

User Action

Either correct the function name to specify an intrinsic supported for this platform, or remove it from the pragma.

UNKMSGCMD, Bad or missing command in pragma message. Pragma is ignored.

Description

The #pragma message directive must be followed by an identifier that specifies message-related action for the compiler to perform. Either something other than an identifier was found, or the identifier did not specify one of the valid actions. The compiler will ignore the pragma.

User Action

Specify a valid action for #pragma message.

UNKMSGID, Unknown message id or group "*id*" is ignored.

Description

A message identifier in a #pragma message directive did not specify a valid message id or message group. The identifier will be ignored.

User Action

Update the identifier so that it specifies a valid message id or message group.

UNKNOWNLINK, The specified linkage is undefined. Pragma is ignored.

Description

The linkage specified in a #pragma use_linkage directive has not been defined by an earlier #pragma linkage directive. The compiler will ignore the entire pragma.

User Action

Either define the linkage first or change the linkage name.

UNKNOWNMACRO, "*name*" is not currently defined as a macro. It has been replaced by the constant zero.

Description

An identifier found in an #if or #elif is not defined. This might not have been what you intended. The compiler will replace the identifier with the constant zero.

User Action

Verify the use of the identifier.

UNKNOWNPRAGMA, The pragma "*pragma text*" is unrecognized.

Description

A pragma that has no meaning to VSI C was encountered. The pragma will be ignored.

User Action

Make sure that you did not misspell the pragma. Also, make certain you are running the correct version of VSI C. If the spelling and compiler version are correct, understand that this pragma will have no effect.

UNKNOWNPRGMA, Unrecognized #pragma directive.

Description

This #pragma preprocessing directive is not recognized by VSI C. The directive will be ignored.

User Action

Make sure that this is the intended behavior.

UNKPSECTATTR, Unknown psect attribute for extern model. Attribute is ignored.

Description

A psect attribute specified in a #pragma extern_model is invalid. In general, the psect attributes accepted by VSI C match those accepted by the assembler. The psect attribute will be ignored.

User Action

Correct the psect attribute.

UNMATCHENDIF, Out of place #endif directive ignored.

Description

An #endif preprocessing directive was encountered without a previous #if directive. The directive will be ignored.

User Action

Remove the directive.

UNNAMEDMEM, An unnamed member does not have a bitfield, struct, or union type. Member is ignored.

Description

An unnamed member of a struct or union type has no meaning unless it is a bitfield or a struct/union type. The compiler will ignore this member.

User Action

If the member is desired, give it a name. Otherwise remove the unnamed member.

UNNAMEPARM, In the definition of the function *name*, a parameter has no name.

Description

This function declaration contained a parameter type but no parameter name.

User Action

Provide a name for the formal parameter.

UNNECCDD, It is not necessary to include this dictionary directive, if other unused dictionary directives and unused include directives are removed.

Description

There is some reference to this file from an unused include file or from an unused dictionary directive when using the current set of compilation options. If you remove the unused include files and unused dictionary directives, this dictionary directive could also be eliminated when compiling with the current set of compilation options.

User Action

When compiling with the current set of compilation options, to increase compilation efficiency you may exclude this dictionary directive if you also remove other unused files.

UNNECINCL, It is not necessary to include this file, if other unused include directives are removed.

Description

There is some reference to this file from another include file or dictionary directive that is not used when using the current set of compilation options. If you remove the unused include files and unused dictionary directives, this include file could also be eliminated when compiling with the current set of compilation options.

User Action

When compiling with the current set of compilation options, to increase compilation efficiency you may exclude this include file if you also remove other unused files.

UNREACHCODE, Code at or just after this location can never be executed*inline info*.

Description

The compiler has detected code that can never be executed. Often unreachable code represents a real coding error such as a label that is incorrectly spelled, or a statement that was inserted on the wrong line. But sometimes it occurs in good code as a result of logical expressions that depend only on the values of constants (typically through macro expansion).

User Action

Usually any code correction is obvious. And often it is straightforward to rewrite compile-time logical expressions in terms of preprocessing constructs to avoid this diagnostic. But in some programs it may be necessary to suppress this informational message explicitly in order to obtain a diagnostic-free compilation of production code, since rewriting the expression not to be evaluated at compile time would impact performance.

UNREFADECL, This local identifier is declared but not referenced in this module.

Description

A declaration was found for an identifier which is not referenced in this module

User Action

Examine your code to determine if this declaration is needed in this module.

UNREFDECL, This identifier is declared but not defined or referenced in this module.

Description

A declaration was found for an identifier which is not defined or referenced in this module

User Action

Examine your code to determine if this declaration is needed in this module.

UNREFLABEL, The user label "*label*" is never referenced.

Description

This user label has been defined, but there are no references to it.

User Action

Remove the label.

UNREFSDECL, A static variable is declared but never referenced in this module.

Description

This identifier is defined but never referenced when using the current set of compilation options.

User Action

Examine your code to determine if this definition is needed in this module.

UNREFSFUNC, A static function definition or prototype is found, but never referenced.

Description

A static function declaration was found in this module, but is unused when compiling with the current settings.

User Action

Examine your code to determine if this function is needed in this module.

UNREFTYP, This type is never referenced in this module.

Description

A type is declared but never referenced when using the current set of compilation options.

User Action

Examine your code to determine if this declaration is needed in this module.

UNRLINKATTR, Unrecognized attribute for linkage pragma. Pragma is ignored.

Description

The compiler encountered an attribute in a #pragma linkage directive that it did not recognize. The message should point to the offending attribute. The compiler will ignore the entire pragma.

User Action

Correct the directive.

UNSIGNEDPRES, *context*the conversion of the unsigned char/short value "*expression*" to unsigned int shows one example of this program's use of unsigned-preserving integral promotion. This differs from the value-preserving semantics of standard C compilers.

Description

This expression shows one of possibly many places where this compilation uses unsigned-preserving semantics for small integer promotions rather than value-preserving semantics required of standard C compilers. In cases where an unsigned char or unsigned short int is promoted to an integer, there are two different ways the convert could happen. Standard C requires that the type be converted to a signed int (value-preserving semantics) while some older compilers will convert to an unsigned int (unsigned-preserving semantics). The difference in the choice of int or unsigned int can have an impact on results of expressions that use the converted value. The compiler cannot determine whether or not a particular instance of this usage will cause an observable behavior difference in the program. For more information, consult Section 3.2.1.1 of the Rationale for ANSI C.

User Action

Be aware that standard compilers might interpret this expression differently.

UNSTRUCTMEM, The declaration of a member that is an unnamed struct or union type is an extension and might not be portable.

Description

VSI C allows a member of a struct or union to be an unnamed struct or union type. This is an extension of VSI C that other compilers might not support. In addition this behavior does not conform to the C standard.

User Action

If portability is desired, provide a name for the struct/union member.

UNSUPCONV, Hexadecimal floating point constants are not yet implemented.

Description

Hexadecimal floating point constants are a new C99 feature that is not yet supported on this platform.

User Action

Please use traditional syntax for floating point numbers.

UNSUPCONVSPEC, *context*this argument to *function name* has a conversion specification "*incorrect conversion*" that is not supported or not fully supported on this platform.

Description

The compiler has detected a conversion specification that will not work as specified on this platform.

User Action

Review the documentation for this function and modify the conversion specification as necessary to accomplish your objective.

UNSUPCONVV, Hexadecimal floating point constants are not supported on this platform.

Description

Hexadecimal floating point constants are a new feature in C99 that is not being implemented on the VAX platform.

User Action

Please use traditional syntax for floating point numbers.

UNSUPIEEE, The _FASTMATH version of this function has been specified, but _FASTMATH routines do not support the IEEE behaviors requested and will simply trap and terminate when given arguments or computing values outside the normal range.

Description

The compiler has recognized a math intrinsic function that has a _FASTMATH version and the compilation has defined the macro _FASTMATH, but command line options have also specified IEEE trapping behaviors other than the default of flushing underflow to zero and aborting on all others.

User Action

If the body of your code relies on IEEE denormals, infinities, or nans, but is careful to condition the arguments to math library functions to avoid passing or computing these values, you may ignore or suppress this warning. Otherwise, you should either remove the options specifying non-default IEEE behavior or else undefine the _FASTMATH macro.

UNSUPPTYPE, The CDD description for *name* specifies a data type not supported in C.

Description

There is no VSI C datatype to exactly represent this type. VSI C has created a declaration of the same total size as the unsupported data type.

User Action

If the type provided by the VSI C compiler is not satisfactory, change the CDD description to one that the compiler can represent more exactly.

UNUSEDCDD, This CDD record appears to be unused.

Description

The contents of this CDD record are not used by the rest of the compilation.

User Action

For compilation efficiency, you can exclude this dictionary directive when compiling with the current set of compilation options.

UNUSEDINCL, This nested include file appears to be unused.

Description

The contents of this include file are not used by the rest of the compilation.

User Action

For compilation efficiency, you can exclude this include file when compiling with the current set of compilation options.

UNUSEDTOP, This include directive does not contribute to the compilation, perhaps because the file has already been included.

Description

The contents of this top-level include file are not used by the rest of the compilation. This message can occur when the include file has already been included, perhaps by a nested include file.

User Action

For compilation efficiency, you can exclude this include file when compiling with the current set of compilation options.

USELESSALIGN, *context* _align cannot be used with the *class* storage class. Modifier _align is ignored.

Description

The storage class modifier _align is meaningless for objects declared with register, globalref, or globalvalue storage class. The compiler ignores the _align.

User Action

Remove the _align storage class modifier.

USELESSSTOMOD, *contextnoshare or readonly* cannot be used with the typedef storage class. Modifier is ignored.

Description

The storage class modifiers noshare and readonly are meaningless for typedefs. The compiler ignores the storage class modifier.

User Action

Remove the storage class modifier.

USELESSTYPED, This typedef declaration is useless because it does not declare a typedef name.

Description

This typedef declaration does not declare a typedef name. This case can occur when a declaration tries to declare both a tag and a typedef, but the name of the typedef is not included.

User Action

Either remove the typedef keyword or add a typedef name.

USELESSTYPEQUAL, *context*this type qualifier will have no effect.

Description

A type qualifier is applied only to the declarators in a declaration. Declarations that lack declarators are permitted if they declare a tag or an enumeration constant, but in such cases type qualifiers are not useful.

User Action

Remove the type qualifier, or change this to a typedef declaration that declares a name for the type and use that typedef name to refer to the qualified type.

VAARGSBODY, __VA_ARGS__ may not appear except in a function-like macro that uses the ellipsis notation in the parameters.

Description

The identifier __VA_ARGS__ may only appear in the replacement list of a function-like macro definion that uses ellipsis notation in the parameters.

User Action

Either remove __VA_ARGS__ or change its spelling.

VAARGSFORMAL, __VA_ARGS__ may not be used as a formal parameter.

Description

The identifier __VA_ARGS__ may only appear in the replacement list of a function-like macro definion that uses ellipsis notation in the parameters.

User Action

Rename the formal parameter.

VALUENOTSUP, *context*the floating-point constant named "*name*" is not supported in "*fpmode*" representation.

Description

The representation of an IEEE Infinity or NaN has no special meaning when used with non-IEEE floating-point operations.

User Action

Replace the named constant with a conventional floating point constant. The HUGE_VAL macros defined by <math.h> may be used in place of IEEE Infinities with any floating-point representation.

VALUEPRES, *context*the conversion of the unsigned char/short value "*expression*" to signed int shows one example of this program's use of value-preserving integral promotion. This differs from the unsigned-preserving semantics of some older C compilers.

Description

This expression shows one of possibly many places where this compilation uses value-preserving semantics for small integer promotions rather than unsigned-preserving semantics used by some older compilers. In cases where an unsigned char or unsigned short int is promoted to an integer, there are two different ways the convert could happen. Standard C requires that the type be converted to a signed int (value-preserving semantics) while some older compilers will convert to an unsigned int (unsigned-preserving semantics). The difference in the choice of int or unsigned int can have an impact on results of expressions that use the converted value. The compiler cannot determine whether or not a particular instance of this usage will cause an observable behavior difference in the program. For more information, consult Section 3.2.1.1 of the Rationale for ANSI C.

User Action

Be aware that older, non-standard compilers might interpret this expression differently.

VARIANTDCL, A declaration of a variant struct or variant union must have a single declarator that is an identifier.

Description

A variant_struct or variant_union member was either not followed by a declarator or followed by more than one declarator. This is not valid.

User Action

Declare the variant_struct or variant_union member with a single identifier.

VARIANTDUP, The anonymous struct or union member "*member name*" duplicates the name of a member in the enclosing struct or union.

Description

As members of an anonymous structure or union are promoted to membership of the enclosing struct/union type, the names of each element of the anonymous struct/union must not match an element name in the enclosing struct/union. This message can also be output when the variant_struct or variant_union syntax is used instead of the anonymous struct/union.

User Action

Choose a new name for either the offending anonymous struct/union member or the matching member of the enclosing type.

VARIANTEXT, variant struct or union is a language extension.

Description

Declaring a member to be a variant_struct or variant_union is a language extension of VSI C. Other C compilers might not successfully compile a program that uses the extension.

User Action

Consider using an anonymous struct or union (one without a tag or declarator) instead: anonymous structs/unions are supported by VSI C and some other vendors' C compilers.

VARIANTTAG, A variant struct or union cannot have a tag.

Description

A variant_struct or variant_union declaration specified a tag name. This is not allowed.

User Action

Either remove the tag or change the declaration to be a regular struct or union instead of a variant struct or union.

VARNOMEM, A variant struct or variant union can occur only as a member of a struct or union.

Description

A declaration contained a variant_struct or variant_union in some place other than a member of a struct or union. This is not valid.

User Action

Correct the offending declaration.

VERTICALSPDIR, Vertical whitespace within pp directive.

Description

Unexpected vertical white space as been encountered within a preprocessing directive.

User Action

Remove the vertical white space from the directive.

VLAEXTENSION, *context*variable length arrays are a new feature in the C99 standard. Other C compilers may not support this extension.

Description

This is a new language feature in the C99 revision of the standard. While having a standard specification for portability, the feature may not yet be available in all of the compilers you use.

User Action

Determine whether or not the use of this feature will cause portability problems for this code.

VOIDRETURN, The function "*name*" has return type void, and so must not contain a return statement with an expression.

Description

The current function was declared with a void return type. The expression specified in the return value will be evaluated but will not be returned to the caller.

User Action

Either change the return type in the function declaration or remove the return value from the return statement.

VOIDRETURN1, The function "*name*" has return type void. The return statement must not specify a return value even if the return expression has void type.

Description

The current function was declared with a void return type. Although some C compilers allow such a function to return a void expression, this is a violation of the C standard and may not be portable.

User Action

Modify the program so that the return statement does not specify a return value.

VOLATILEFUNC, Ignoring volatile type qualifier in declaration of *name*.

Description

The volatile type qualifier cannot be used with a function type. The compiler will ignore the type qualifier.

User Action

Remove the type qualifier.

WCHARCAT, A character string literal was concatenated with a wide string literal.

Description

The C99 standard defines the behavior of adjacent string concatenation between character string literals and wide string literals, basically promoting the character string to a wide string before forming the wide string result. The older C90 standard gave this construct undefined behavior - it only defined concatenation between adjacent strings of the same kind (all character or all wide). Although this version of VSI C always gives the C99 behavior with diagnostics optional, some compilers (including previous versions of VSI C) may give more severe diagnostics and/or different behaviors.

User Action

Be aware of this if you wish to port the program.

WRTINNOWRT, Writable variable resides in nowrt extern model.

Description

The current extern model places all external objects in a read-only section. An object without a const type qualifier in such a section means that while the compiler will not diagnose writes to the object, any attempt to modify the object at runtime will cause the program to fail. This might not have been what you intended.

User Action

Place non-const objects in sections that can be modified.

XFERINTOVLA, This statement performs an invalid transfer into a block that declares a variably modified type or object. The identifier "*name*" is variably modified, and declared at *where*.

Description

It is invalid to transfer control into a block after that block declares a variably modified type.

User Action

Either remove the transfer, or move the declaration of the variably modified type.

XTRALARGE, Line number is greater than the 32767 specified by the C standard and might not be portable.

Description

A #line preprocessing directive specified a line value that is greater than 32767. While the value is supported by VSI C, the C89 standard specifies that the value must not be greater than 32767. Therefore, this program does not conform to the C89 standard, and the directive might not be accepted by other C compilers.

User Action

Be aware of this if you wish to port the program.

ZERODIV, *context*division by zero occurs in evaluating the expression "*expression*".

Description

A divide by zero occurred while evaluating a constant expression. The value of the expression is undefined.

User Action

Correct the constant expression so that it does not contain a division by zero.

ZERODIVIDE, Division by zero in expression.

Description

A divide by zero occurs in a preprocessor constant expression. The result of the divide will be zero.

User Action

Correct the preprocessor constant expression.

ZEROELEMENTS, *context*zero cannot be used as an element count specifier. The specifier will be ignored, (leaving the member/parameter with an incomplete array type) in this context.

Description

The C standard states that if an element count specifier is a constant expression then it shall have a value greater than zero. For compatibility with some other C compilers, VSI C will accept a zero element count specifier. When appearing in a struct/union member or a parameter, the specifier will be ignored.

User Action

Remove the zero.

ZEROELEMENTS1, *context*zero cannot be used as an element count specifier. It will be replaced with the constant one in this context.

Description

The C standard states that if an element count specifier is a constant expression then it shall have a value greater than zero. For compatibility with some other C compilers, VSI C will accept a zero element count specifier. When appearing outside a struct/union member or a parameter, the compiler will replace the zero with the value one. This may or may not be compatible with the behavior of other C compilers.

User Action

Use a valid element count specifier.

Appendix E. VSI C Limits

The <float.h> and <limits.h> header files define several macros that expand to various implementation-specific limits and parameters. This appendix contains the contents of these header files for VSI C for OpenVMS systems.

E.1. Contents of <float.h>

The <float.h> header file has the following contents:

```
#ifndef __FLOAT_LOADED
#define __FLOAT_LOADED 1
/
*****************************************************************************
**
**  <float.h> - Characteristics of floating types
**
*****************************************************************************
**  Header introduced by the ANSI C Standard
*****************************************************************************
**
** Copyright 2001, 2004 Hewlett-Packard Development Company, L.P.
**
** Confidential computer software. Valid license from HP required for
** possession, use or copying. Consistent with FAR 12.211 and 12.212,
** Commercial Computer Software, Computer Software Documentation, and
** Technical Data for Commercial Items are licensed to the U.S.
** Government under vendor's standard commercial license.
**
*****************************************************************************
*/

#include <decc$types.h>
#pragma   __nostandard
#ifdef __cplusplus
    extern "C" {
#endif


/*
**  The following literals and routines are available on OpenVMS for
**  Alpha, but only after OpenVMS V7.1 or with C++.
*/
#if defined __ALPHA && !defined _ANSI_C_SOURCE
#   if (defined(__DECCXX) || (__CRTL_VER >= 70100000))

    /*
    **  Values for the IEEE Rounding Modes (IEEE ANSI Values)
    **
    **  RZ = Round toward zero (chopped)
    **  RN = Round toward nearest (default, normal)
    **  RP = Round toward plus infinity
    **  RM = Round toward minus infinity
    */
#   define FP_RND_RZ    0
#   define FP_RND_RN    1
```

```
#   define FP_RND_RP    2
#   define FP_RND_RM    3


    /*
    ** IEEE Constants
    */
#   ifdef _IEEE_FP
#      pragma __extern_model __save
#      pragma __extern_model __strict_refdef
       extern double decc$gt_dinfinity;
       extern double decc$gt_dqnan;
       extern double decc$gt_dsnan;
       extern float decc$gs_sinfinity;
       extern float decc$gs_sqnan;
       extern float decc$gs_ssnan;
#      if __X_FLOAT
#         if (__CRTL_VER >= 60200000)
             extern long double decc$gx_long_dbl_infinity;
#         endif
          extern long double decc$gx_long_dbl_qnan;
          extern long double decc$gx_long_dbl_snan;
#      endif
#      pragma __extern_model __restore
#      define DBL_INFINITY decc$gt_dinfinity
#      define LDBL_INFINITY DBL_INFINITY
#      define DBL_QNAN decc$gt_dqnan
#      define DBL_SNAN decc$gt_dsnan
#      define FLT_INFINITY decc$gs_sinfinity
#      define FLT_QNAN decc$gs_sqnan
#      define FLT_SNAN decc$gs_ssnan
#      if __X_FLOAT
#         if (__CRTL_VER >= 60200000)
#           define LDBL_INFINITY decc$gx_long_dbl_infinity
#         else
#           define LDBL_INFINITY DBL_INFINITY
#         endif
#         define LDBL_QNAN  decc$gx_long_dbl_qnan
#         define LDBL_SNAN  decc$gx_long_dbl_snan
#      else
#          define LDBL_INFINITY DBL_INFINITY
#          define LDBL_QNAN DBL_QNAN
#          define LDBL_SNAN DBL_SNAN
#      endif
#   endif
    /*
    ** Macros to get decc$ names
    */
#   if (__CRTL_VER < 70100000)
#      define write_rnd(__p1) decc$write_rnd(__p1)
#      define read_rnd decc$read_rnd
#   endif


    /*
    ** Functions to read and write floating point rounding mode
    */
    unsigned int write_rnd(unsigned int __rnd);
```

```
    unsigned int read_rnd(void);

#   endif
#endif


/*
**  Rounding mode for floating point addition:
*/
#ifdef __BIASED_FLT_ROUNDS
#  define FLT_ROUNDS (__BIASED_FLT_ROUNDS-1) /* use compiler generated
                                                value, if
 present */
#else
#  define FLT_ROUNDS 1
#endif


/*
**  Radix of exponent representation:
*/
#define FLT_RADIX 2


/*
**  Number of FLT_RADIX digits in the mantissa including the hidden bit:
*/
#define __F_FLT_MANT_DIG    24
#define __G_DBL_MANT_DIG    53
#ifdef __ALPHA
#define __S_FLT_MANT_DIG    24
#define __T_FLT_MANT_DIG    53
#define __X_FLT_MANT_DIG    113
#endif


/*
**  Number of decimal digits of precision:
*/
#define __F_FLT_DIG         6
#define __G_FLT_DIG         15
#ifdef __ALPHA
#define __S_FLT_DIG         6
#define __T_FLT_DIG         15
#define __X_FLT_DIG         33
#endif


/*
**  Minimum negative integer such that FLT_RADIX raised to that power
**  minus 1 is a normalized floating-point number:
*/
#define __F_FLT_MIN_EXP     (-127)
#define __G_FLT_MIN_EXP     (-1023)
#ifdef __ALPHA
#define __S_FLT_MIN_EXP     (-125)
#define __T_FLT_MIN_EXP     (-1021)
#define __X_FLT_MIN_EXP     (-16381)
```

```
#endif


/*
**  Minimum negative integer such that 10 raised to that power is in the
**  range of normalized floating-point numbers:
*/
#define __F_FLT_MIN_10_EXP  (-38)
#define __G_FLT_MIN_10_EXP  (-308)
#ifdef __ALPHA
#define __S_FLT_MIN_10_EXP  (-37)
#define __T_FLT_MIN_10_EXP  (-307)
#define __X_FLT_MIN_10_EXP  (-4931)
#endif


/*
**  Maximum integer such that FLT_RADIX raised to that power minus 1 is a
**  representable finite floating point number:
*/
#define __F_FLT_MAX_EXP     127
#define __G_FLT_MAX_EXP     1023
#ifdef __ALPHA
#define __S_FLT_MAX_EXP     128
#define __T_FLT_MAX_EXP     1024
#define __X_FLT_MAX_EXP     16384
#endif


/*
**  Maximum integer such that 10 raised to that power is in the range of
**  representable finite floating-point numbers:
*/
#define __F_FLT_MAX_10_EXP  38
#define __G_FLT_MAX_10_EXP  307
#ifdef __ALPHA
#define __S_FLT_MAX_10_EXP  38
#define __T_FLT_MAX_10_EXP  308
#define __X_FLT_MAX_10_EXP  4932
#endif


/*
**  Maximum representable finite floating-point number:
*/
#define __F_FLT_MAX         1.7014117e+38f
#define __G_FLT_MAX         8.98846567431157854e+307
#ifdef __ALPHA
#define __S_FLT_MAX         3.40282347e+38f
#define __T_FLT_MAX         1.79769313486231570e+308
#define __X_FLT_MAX
 1.18973149535723176508575932662800701619647 7e4932l
#endif


/*
**  The difference between 1.0 and the least value greater than 1.0 that
**  is representable in the given floating-point type
```

```
**    (i.e. 1.0 + epsilon != 1.0):
*/
#define __F_FLT_EPSILON     ((float)(1.0 / (1 << 23)))
#define __G_FLT_EPSILON     (1.0 / (1 << 30) / (1 << 22))
#ifdef __ALPHA
#define __S_FLT_EPSILON     1.19209290e-07f
#define __T_FLT_EPSILON     2.220460492503131e-16
#define __X_FLT_EPSILON     1.92592994438723585305597794258492731853 81e-34l
#endif


/*
**  Minimum normalized positive floating-point number:
*/
#define __F_FLT_MIN         ((float) 2.9387358770557187 7e-39)
#define __G_FLT_MIN         5.562684646268003 46e-309
#ifdef __ALPHA
#define __S_FLT_MIN         1.17549435e-38f
#define __T_FLT_MIN         2.2250738585072014e-308
#define __X_FLT_MIN     ((long double)

 3.36210314311209350626267781732175260259 81e-4932l)
#endif


/*
**  Define the FLT values to be either the __S or __F values based on IEEE
*/
#if __IEEE_FLOAT
#   define FLT_MANT_DIG     __S_FLT_MANT_DIG
#   define FLT_DIG          __S_FLT_DIG
#   define FLT_MIN_EXP      __S_FLT_MIN_EXP
#   define FLT_MIN_10_EXP   __S_FLT_MIN_10_EXP
#   define FLT_MAX_EXP      __S_FLT_MAX_EXP
#   define FLT_MAX_10_EXP   __S_FLT_MAX_10_EXP
#   define FLT_MAX          __S_FLT_MAX
#   define FLT_EPSILON      __S_FLT_EPSILON
#   define FLT_MIN          __S_FLT_MIN
#else
#   define FLT_MANT_DIG     __F_FLT_MANT_DIG
#   define FLT_DIG          __F_FLT_DIG
#   define FLT_MIN_EXP      __F_FLT_MIN_EXP
#   define FLT_MIN_10_EXP   __F_FLT_MIN_10_EXP
#   define FLT_MAX_EXP      __F_FLT_MAX_EXP
#   define FLT_MAX_10_EXP   __F_FLT_MAX_10_EXP
#   define FLT_MAX          __F_FLT_MAX
#   define FLT_EPSILON      __F_FLT_EPSILON
#   define FLT_MIN          __F_FLT_MIN
#endif


/*
**  Define the DBL values to be either the __S or __F values based on IEEE
*/
#if __IEEE_FLOAT
#   define DBL_MANT_DIG     __T_FLT_MANT_DIG
#   define DBL_DIG          __T_FLT_DIG
#   define DBL_MIN_EXP      __T_FLT_MIN_EXP
```

```
#   define DBL_MIN_10_EXP   __T_FLT_MIN_10_EXP
#   define DBL_MAX_EXP      __T_FLT_MAX_EXP
#   define DBL_MAX_10_EXP   __T_FLT_MAX_10_EXP
#   define DBL_MIN          __T_FLT_MIN
#elif __G_FLOAT
#   define DBL_MANT_DIG     __G_DBL_MANT_DIG
#   define DBL_DIG          __G_FLT_DIG
#   define DBL_MIN_EXP      __G_FLT_MIN_EXP
#   define DBL_MIN_10_EXP   __G_FLT_MIN_10_EXP
#   define DBL_MAX_EXP      __G_FLT_MAX_EXP
#   define DBL_MAX_10_EXP   __G_FLT_MAX_10_EXP
#   define DBL_MIN          __G_FLT_MIN
#else
#   define DBL_MANT_DIG     56
#   define DBL_DIG          16
#   define DBL_MIN_EXP      __F_FLT_MIN_EXP
#   define DBL_MIN_10_EXP   __F_FLT_MIN_10_EXP
#   define DBL_MAX_EXP      __F_FLT_MAX_EXP
#   define DBL_MAX_10_EXP   __F_FLT_MAX_10_EXP
#   define DBL_MIN          2.9387358705571877e-39
#endif


#if __IEEE_FLOAT
#   define DBL_MAX          __T_FLT_MAX
#elif __G_FLOAT
#   define DBL_MAX          __G_FLT_MAX
#else
#   ifndef __ALPHA
#       define DBL_MAX      1.70141183460469229e+38
#   else
#       define DBL_MAX      1.70141183460469213e+38
#   endif
#endif


#if __IEEE_FLOAT
#   define DBL_EPSILON      __T_FLT_EPSILON
#elif __G_FLOAT || (__D_FLOAT && defined(__ALPHA))
#   define DBL_EPSILON      (1.0 / (1 << 20) / (1 << 16) / (1 << 16))
#else
#   define DBL_EPSILON      (1.0 / (1 << 23) / (1 << 16) / (1 << 16))
#endif


/*
**  Define the LDBL values based on __X_FLOAT
*/
#if __X_FLOAT
#   define LDBL_MANT_DIG    __X_FLT_MANT_DIG
#   define LDBL_DIG         __X_FLT_DIG
#   define LDBL_MIN_EXP     __X_FLT_MIN_EXP
#   define LDBL_MIN_10_EXP  __X_FLT_MIN_10_EXP
#   define LDBL_MAX_EXP     __X_FLT_MAX_EXP
#   define LDBL_MAX_10_EXP  __X_FLT_MAX_10_EXP
#   define LDBL_MAX         __X_FLT_MAX
#   define LDBL_EPSILON     __X_FLT_EPSILON
```

```
#   define LDBL_MIN
 3.362103143112093506262677817321752602598 1e-4932l
#else
#   define LDBL_MANT_DIG    DBL_MANT_DIG
#   define LDBL_DIG         DBL_DIG
#   define LDBL_MIN_EXP     DBL_MIN_EXP
#   define LDBL_MIN_10_EXP  DBL_MIN_10_EXP
#   define LDBL_MAX_EXP     DBL_MAX_EXP
#   define LDBL_MAX_10_EXP  DBL_MAX_10_EXP
#   define LDBL_MAX         DBL_MAX
#   define LDBL_EPSILON     DBL_EPSILON
#   define LDBL_MIN         DBL_MIN
#endif

#ifdef __cplusplus
    }
#endif

#pragma __standard
#endif /* __FLOAT_LOADED */
```

E.2. Contents of <limits.h>

The `<limits.h>` header file has the following contents:

```
#ifndef __LIMITS_LOADED
#define __LIMITS_LOADED 1
/
*****************************************************************************
**
**  <limits.h> - Sizes of integral types
**
*****************************************************************************
**  Header introduced by the ANSI C Standard
*****************************************************************************
**
** Copyright 2001, 2004 Hewlett-Packard Development Company, L.P.
**
** Confidential computer software. Valid license from HP required for
** possession, use or copying. Consistent with FAR 12.211 and 12.212,
** Commercial Computer Software, Computer Software Documentation, and
** Technical Data for Commercial Items are licensed to the U.S.
** Government under vendor's standard commercial license.
**
*****************************************************************************
**                                    Note
*****************************************************************************
**
**  Section 2.2.4.2 of the Rationale states "The limits for the maxima and
**  minima of unsigned types are specified as unsigned constants..."
**
**  The alert reader will notice there are no minima for the unsigned
**  types, but we will follow the Rationale's advice anyway.
**
*****************************************************************************
**                               Implementors Note
*****************************************************************************
```

```
**
**  Some constants in this file such as INT_MIN is defined in terms of an
**  expression involving an INT_MAX which is a constant value.  Please do
**  not be tempted to speed processing up by evaluating those expressions
**  into constant values.  This will cause things to not work correctly.
*****************************************************************************
*/

#include <decc$types.h>
#pragma   __nostandard

/*
**  Number of bits for the smallest object that is not a bit-field (byte)
*/
#define   CHAR_BIT    8


/*
**  Minimum and maximum values for "signed/unsigned char"
*/
#define   UCHAR_MAX   255u
#define   SCHAR_MAX   127
#define   SCHAR_MIN   (-SCHAR_MAX - 1)


/*
**  Minimum and maximum values for "char" affected by /unsigned_char
 qualifier
*/
#ifdef __UNSIGNED_CHAR
#define   CHAR_MIN    0
#define   CHAR_MAX    UCHAR_MAX
#else
#define   CHAR_MIN    SCHAR_MIN
#define   CHAR_MAX    SCHAR_MAX
#endif


/*
**  Minimum and maximum values for "signed/unsigned short int"
*/
#define   USHRT_MAX   65535u
#define   SHRT_MAX    32767
#define   SHRT_MIN    (-SHRT_MAX - 1)


/*
**  Minimum and maximum values for "signed/unsigned int"
*/
#define   UINT_MAX    4294967295u
#define   INT_MAX     2147483647
#define   INT_MIN     (-INT_MAX - 1)


/*
**  Minimum and maximum values for "signed/unsigned long int"
*/
#define   ULONG_MAX   4294967295u
```

```
#define   LONG_MAX    2147483647
#define   LONG_MIN    (-LONG_MAX - 1)


/*
**  Minimum and maximum values for "signed/unsigned __intxx"
*/
#define __UINT16_MAX  65535u
#define __INT16_MAX   32767
#define __INT16_MIN   (-__INT16_MAX - 1)

#define __UINT32_MAX  4294967295u
#define __INT32_MAX   2147483647
#define __INT32_MIN   (-__INT32_MAX - 1)


#ifdef __ALPHA
#define __UINT64_MAX  18446744073709551615u
#define __INT64_MAX   9223372036854775807
#define __INT64_MIN   (-__INT64_MAX - 1)
#endif

#if __CRTL_VER < 60200000
#   define  MB_LEN_MAX              1   /*  Before OpenVMS V6.2  */
#else
#   define  MB_LEN_MAX              8   /*  After  OpenVMS V6.2  */
#endif

/*
**  Limits which changed beginning with OpenVMS V6.2
*/
#   if defined(_XOPEN_SOURCE) || !defined(_ANSI_C_SOURCE)
#   define  COLL_WEIGHTS_MAX         5   /*  Max collate weights  */
#   define  NL_TEXTMAX            8192
#   define  NL_SETMAX            65535
#   define  NL_MSGMAX            65535
#   define  CHARCLASS_NAME_MAX      14
#   define  NL_ARGMAX                9
#   define  NL_LANGMAX              14
#   define  TZNAME_MAX              15
#   define  SSIZE_MAX          INT_MAX

    /*
    ** Limits needed to support *conf() functions.
    */
#   define  BC_BASE_MAX    -1    /* Max ibase and obase values
                                ** for bc not implemented */
#   define  BC_DIM_MAX     -1    /* Max num elements in array
                                ** for bc not implemented */
#   define  BC_SCALE_MAX   -1    /* Max scale value allowed by
                                ** bc not implemented */
#   define  BC_STRING_MAX  -1    /* Max len of string constant
                                ** by bc not implemented */

#   define  EXPR_NEST_MAX  (-1)  /* Max num expression nested
                                   for expr */
#   define  LINE_MAX       (-1)  /* Max len of utility input
                                ** line */
```

```
#   define  RE_DUP_MAX     (-1)  /* Max num repeated reg for
                                   ** interval */

#   define  NGROUPS_MAX    0     /* User can be in no extra groups */
#   define  PASS_MAX       31    /* Max bytes in a password */
#   define  ARG_MAX        4096  /* Max len of arg to exec rtns */

    /*
    **  These are used by pathconf() as well as others
    */
#   define  LINK_MAX   1      /* Only 1 link to a file */
#   define  MAX_CANON  511    /* Max bytes in terminal canonical
                                ** input */
#   define  MAX_INPUT  511    /* Max bytes required as input
                                ** before reading */
#   define  NAME_MAX   255    /* Max bytes in filename */
#   define  PATH_MAX   255    /* Max bytes in pathname */
#   define  PIPE_BUF   512    /* Max atomic bytes on write to pipe */

    /*
    **  New limits with DEC C V5.2
    */
#   define _POSIX_PIPE_BUF            512

#endif /* XOPEN_SOURCE */

#if defined(_XOPEN_SOURCE_EXTENDED) || !defined(_ANSI_C_SOURCE)
#   define  ATEXIT_MAX   32767 /* Max number of functions that
                                 ** may be registered with atexit().
                                 ** essentially unlimited
                                 */
#   define  IOV_MAX      (-1)  /* Maximum number of iovec
                                 ** structures that one process
                                 ** has available for use with
                                 ** readv() or writev() */
#endif



/*
**  Macros defined by the POSIX 1003.1c-1995 formally approved at
**  the June 1995 meeting of the IEEE Standards Board.  The correct
**  feature test macro for strictly conforming POSIX 1003.1c-1995
**  applications is:
**
**      #define _POSIX_C_SOURCE 199506L
*/
#if _POSIX_C_SOURCE >= 199506 || !defined _ANSI_C_SOURCE

#   ifndef _POSIX_THREAD_DESTRUCTOR_ITERATIONS
#      define _POSIX_THREAD_DESTRUCTOR_ITERATIONS   4
#   endif

#   ifndef _POSIX_THREAD_KEYS_MAX
#      define _POSIX_THREAD_KEYS_MAX                128
#   endif

#   ifndef _POSIX_THREAD_THREADS_MAX
```

```
#       define _POSIX_THREAD_THREADS_MAX             64
#   endif

#   ifndef PTHREAD_DESTRUCTOR_ITERATIONS
#       define PTHREAD_DESTRUCTOR_ITERATIONS
                                _POSIX_THREAD_DESTRUCTOR_ITERATIONS
#   endif

#   ifndef PTHREAD_KEYS_MAX
#       define PTHREAD_KEYS_MAX              255
#   endif

#   ifndef PTHREAD_STACK_MIN
#       if defined __ALPHA
#          define PTHREAD_STACK_MIN                  8192
#       else
#          define PTHREAD_STACK_MIN                  1024
#       endif
#   endif

#endif /* _POSIX_C_SOURCE >= 199506 */

#pragma __standard

#endif /* __LIMITS_LOADED */
```

VSI C Glossary

additive operator	An operator that performs addition (+) or subtraction (–). These operators perform arithmetic conversion on each of the operands, if necessary. See Also *arithmetic conversion rules*.
aggregate	A data structure (array, structure, or union) composed of segments called members. You declare the members to be of either a scalar or aggregate data type. Members of an array are called elements and must be of the same data type. A structure has named members that can be of different data types. A union is a structure that is as long as its longest declared member and that contains the value of only one member at a time.
ampersand (&)	As a unary operator, computes the address of its operand. As a binary operator, performs a bitwise AND on two operands; both must be of an integral type. As an assignment operator (&=), performs a bitwise AND on two expressions and assigns the result to the left object. The double ampersand (&&), a binary operator, performs a logical AND on two operands. See Also *binary operator*, *bitwise operator*, *logical operator*, *unary operator*.
argument	An expression that appears within the parentheses of a function call. The expression is evaluated and the result is copied into the corresponding parameter of the called function. and . See Also *parameter*, *argument passing*.
argument passing	The mechanism by which the value of the argument in a function call is copied to a parameter in the called function. In C, all arguments are passed by value; that is, the parameter receives a copy of the argument's value. Therefore, a function called in C cannot modify the value of an argument except by using its address. In general, addresses are passed using the ampersand operator (&) in the function call or by passing a pointer variable. In addition, using an array or function name (an array with no brackets or function identifier with no parentheses) as an argument results in the passing of the address of the array or function. See Also *ampersand (&)*.
arithmetic conversion rules	The set of rules that govern the changing of a value of an operand from one data type to another in arithmetic expressions. Conversions take place in assignments by changing the type of the right operand's result to that of the object referred to by the left operand; the resultant type also applies to the assignment expression. Conversions are also performed when arguments are passed to functions.
arithmetic operator	A C operator that performs a mathematical operation. In an expression, certain operations take precedence (are performed first) over other operations. The unary minus operator (–) is at the highest level of precedence. At the next level are the binary operators for multiplication

	(*), division (/), and mod (%). At the next level are addition (+) and subtraction (–). There is no unary plus operator, and there is no exponentiation operator. If necessary, all the binary operators perform the arithmetic conversions on their operands. See Also *arithmetic conversion rules*.
arithmetic type	One of the integral data types, enumerated types, `float`, or `double`.)
array	An aggregate data type consisting of subscripted members, called elements, all of the same type. Elements of an array can be one of the fundamental types or can be structures, unions, or other arrays (to form multidimensional arrays).
assignment expression	An expression that has the following form: `E1 asgnop E2`. Expression E1 must evaluate to an lvalue, the asgnop operator is an assignment operator, and E2 is an expression. The type of an assignment expression is that of its left operand. The value of an assignment expression is that of the left operand after the assignment takes place. If the operator is of the form op=, then the operation E1 op (E2) is performed, and the result is assigned to the object referred to by E1; E1 is evaluated once.
assignment operator	The combination of an arithmetic or bitwise operator with the assignment symbol (=); also, the assignment symbol by itself. See Also *assignment expression*.
asterisk (*)	As a unary operator, treats its operand as an address and results in the contents of that address. As a binary operator, multiplies two operands, performing the arithmetic conversions, if necessary. As an assignment operator (*=), multiplies an expression by the value of the object referred to by the left operand, and assigns the product to that object. See Also *binary operator*, *unary operator*.
binary operator	An operator that is placed between two operands. The binary operators include arithmetic operators, shift operators, relational operators, equality operators, bitwise operators (AND, OR, and XOR), logical connectives, and the comma operator, in that order of precedence. All binary operators group from left to right. VSI C has no exponentiation operator. The `exp` library function must be used instead.
bitwise operator	An operator that performs Boolean algebra on the binary values of two operands, which must be integral. If necessary, the operators perform the arithmetic conversions. Both operands are evaluated. All bitwise operators are associative, and expressions using them may be rearranged. The operators include, in order of precedence, the single ampersand (&) (bitwise AND), the circumflex (^) (bitwise exclusive OR), and the single bar (|) (bitwise inclusive OR).
block	See *compound statement*.
block activation	The run-time activation of a block or function, in which local `auto` and `register` variables are allocated storage and, if they are

declared with initializers, given initial values. Variables of storage class `static`, `extern`, `globaldef`, and `globalvalue` are allocated and initialized at link time. The block activation precedes the execution of any executable statements in the function or block. Functions are activated when they are called. Internal blocks (compound statements) are activated when the program control flows into them. Internal blocks are not activated if they are entered by a `goto` statement, unless the `goto` target is the label of the block rather than the label of some statement within the block. If a block is entered by a `goto` statement, references to `auto` and `register` variables declared in the block are still valid references, but the variables may not be properly initialized. Blocks that make up the body of a `switch` statement are not activated; `auto` or `register` variables declared in the block are not initialized.

built-in functions

The function definitions that are part of the VSI C compiler for OpenVMS systems. A call to one of these functions does not call a function in a run-time library or in your program. Most of the built-in functions access the VAX hardware instructions to perform operations quickly that are cumbersome, slow, or impossible in the C language.

cast

An expression preceded by a cast operator of the form (type_name). The cast operator forces the conversion of the evaluated expression to the given type. The expression is assigned to a variable of the specified type, which is then used in place of the whole construction. The cast operator has the same precedence as the other unary operators.

CDD/Repository

An optional OpenVMS software product, available under a separate license, that maintains a set of data structure definitions that many programs on a system, written in many languages, can access. The language-independent definitions are translated into the target language when they are included in the program stream. You can include the CDD records in VSI C programs using the `#dictionary` preprocessor directive. This directive is specific to VSI C for OpenVMS systems, and is not portable.

character

- A member of the ASCII character set.
- An object of type `char`, which is stored in a single byte of memory. An object of type `char` always represents a single character, not a string.
- A constant of type `char` consisting of up to four ASCII characters enclosed in apostrophes (’ ’) not quotation marks (" ").

See Also *string*.

comma operator

A C operator used to separate two expressions as follows: `E1, E2`.

The expressions E1 and E2 are evaluated left to right, and the value of E1 is discarded. The type and value of the comma expression are those of E2.

comment

A sequence of characters introduced by the pair /* and terminated by */. Comments are ignored during compilation. They may not be nested.

Common Data Dictionary (CDD)	See *CDD/Repository*.
compilation unit	All the source files compiled to form a single object module. In other C documentation, the term source file is synonymous with the OpenVMS compilation unit, which is not necessarily a single source file. Declarations and definitions within a compilation unit determine the lexical scope of functions and variables.
compound statement	Valid C statements enclosed in braces ({ }). Compound statements can also include declarations. The scope of these variables is local to the compound statement. A compound statement, when it is not the body of a function, is called a block.
conditional operator	The C operator (?:), which is used in conditional expressions of the following form: `E1 ? E2 : E3`. E1, E2, and E3 are valid C expressions. E1 is evaluated, and if it is nonzero, the result is the value of E2; otherwise, the result is the value of E3. Either E2 or E3 is evaluated, but not both.
constant	A primary expression whose value does not change. A constant may be literal or symbolic.
constant expression	An expression involving only constants. Constant expressions are evaluated at compile time so they may be used wherever a constant is valid.
conversion	The changing of a value from one data type to another. Conversions take place in assignments by changing the type of the right operand's result to that of the object referred to by the left operand; the resultant type also applies to the assignment expression. Conversions are also performed when arguments are passed to functions: `char` and `short` become `int`; `unsigned char` and `unsigned short` become `unsigned int` if no function prototype is in scope; `float` becomes `double`. Conversions can also be forced by means of a cast. Conversions are performed on operands in arithmetic expressions by the arithmetic conversions.
conversion characters	A character used with the C RTL Standard I/O functions that is preceded by a percent sign (%) and specifies an input or output format. For example, letter d instructs the function to input/output the value in a decimal format.
Curses	A screen management package comprised of C RTL functions and macros that create and modify defined sections of the terminal screen, and optimize cursor movement. Curses defines rectangular regions on the terminal display that you may write upon, rearrange, move to new positions on the screen, and delete from the screen. These rectangular regions are called windows. To use any of the Curses functions or macros, you must include the `<curses.h>` header file using the `#include` preprocessor directive.
data definition	The syntax that both declares the data type of an object and reserves its storage. For variables that are internal to a function, the data definition

	is the same as the declaration. For external variables, the data definition is external to any function (an external data definition).
data-type modifier	Keywords that affect the allocation or access of data storage. The two data-type modifiers are `const` and `volatile`.
declaration	A statement that gives the data type and possibly the storage class of one or more variables.
DEC/Shell	An optional OpenVMS software product available under a separate license that is a command-language interpreter based on the UNIX Version 7.0 Bourne Shell with commands for interactive program development, device and data file manipulation, and interactive and batch execution. DEC/Shell RTL functions were added to the C RTL so that valid DEC/Shell file specifications could be used in VSI C for OpenVMS source programs. See Also *file specification*.
dictionaries	A hierarchical organization, similar to the organization of directories and subdirectories, of data structure definitions in the CDD/Repository. See Also *CDD/Repository*.
directives	See *preprocessor directives*.
elements	Members of an array. See Also *aggregate*.
enumerated type	A type defined (with the `enum` keyword) to have an ordered set of integer values. The integer values are associated with constant identifiers named in the declaration. Although `enum` variables are stored internally as integers, use them in programs as if they have a distinct data type named in the `enum` declaration.)
equality operator	One of the operators equal to (==) or not equal to (!=). They are similar to the relational operators, but at the next lower level of precedence.
exponentiation operator	The C language does not have an exponentiation operator. Use the C RTL function `exp`.
expression	A series of characters that the compiler can use to produce a value. Expressions have one or more operands and, usually, one or more operators. An identifier with no operator is an expression that yields a value directly. Operands are either identifiers (such as variable names) or other expressions, which are sometimes called subexpressions. See Also *operator*, *macro*.
external storage class	A storage class that permits identifiers to have a link-time scope that can possibly span object modules. Identifiers of this storage class are defined outside of functions using no storage-class specifier, and are declared, optionally, throughout the program using the `extern` specifier. External variables provide a means other than argument passing for exchanging data between the functions that comprise a C program. See Also *link-time scope*.

file descriptor	In the UNIX environment, the integer that identifies a file.
file specification	An identifier that specifies an existing file. There are two types of valid file specifications in VSI C: OpenVMS specifications and DEC/Shell specifications. DEC/Shell specifications are a subset of UNIX specifications.)
floating type	One of the data types `float` or `double`, representing a single- or double-precision, floating-point number. There are two implementations of the data type `double`: D_floating and G_floating. The range of values for the D_floating variables is the same as that for `float` variables, but the precision is 16 decimal digits, as opposed to 7. Programs that use G_floating variables must use the /FLOAT=G_FLOAT (or /G_FLOAT) command-line qualifier. A G_floating variable has considerably greater range, but has less precision.)
function	The primary unit from which C programs are constructed. A function definition begins with a name and parameter list, followed by the declarations of the parameters (if any) and the body of the function enclosed in braces ({ }). The function body consists of the declarations of any local variables and the set of statements that perform its action. Functions do not have to return a value to the caller. All C functions are external; that is, a function may not contain another function. See Also *function call*.
function call	A primary expression, usually a function identifier followed by parentheses, that is used to invoke the function. The parentheses contain a (possibly empty) comma-separated list of expressions that are the arguments to the function. Any previously undeclared identifier followed immediately by parentheses is declared as a function returning `int`. Any function may call itself recursively.
function inline expansion	A replacement of a function call with code that performs the actions of the defined function. This process reduces execution time. By default, VSI C attempts to expand inline all functions. You can use the `#pragma inline` directive to provide inline expansion for functions that VSI C does not expand inline by default. See Also *pragma*.
function unrolling	See *function inline expansion*.
fundamental type	The set of arithmetic data types plus pointers. In general, the fundamental types comprise those data types that can be represented naturally on a VAX processor; usually, this means integers and floating-point numbers of various machine-dependent sizes, and machine addresses.
global storage class	A storage class that permits identifiers to have a link-time scope that can possibly span object modules. Identifiers of this storage class are defined using the `globaldef` storage-class specifier, and are declared, optionally, throughout the program using the `globalref` specifier. You can use the `globalvalue` specifier to define a global symbol, or constant. Global variables provide a means other than

	argument passing for exchanging data between the functions that comprise a VSI C program. See Also *link-time scope*.
identifier	A sequence of letters and digits, the first 255 of which must be unique. The underscore (_) and dollar sign ($) are letters in this context. The first character of an identifier must be a letter. Upper- and lowercase letters specify different identifiers in VSI C. However, all external names are converted to uppercase to be consistent with the OpenVMS environment and are only 31 characters in length.
initializer	The part of a declaration that gives the initial value(s) for the preceding declarator. An initializer consists of an equal sign (=) followed by either a single expression or a comma-separated list of one or more expressions in braces.
inline expansion	See *function inline expansion*.
integral type	One of the data types `char` or `int` (all sizes, signed or unsigned).
internal storage class	A storage class that permits identifiers declared inside of a function body to be recognized only from the declaration to the end of the immediately enclosing block. Identifiers of the internal storage class are declared using the `auto` and `register` storage-class specifiers. See Also *scope*.
keyword	A character string that is reserved by the C language and cannot be used as an identifier. Keywords identify statements, storage classes, data types, and the like. Library function names are not C keywords; you may redefine function names.)
lexical scope	The area in which the compiler recognizes a declared identifier within a given compilation unit. See Also *scope*.
License Management Facility (LMF)	A process by which you register and use some OpenVMS software products. See your VSI C installation guide for more information.
lifetime	The length of time for which storage for a variable is allocated. See Also *external storage class*, *internal storage class*, *program section (psect)*.
link libraries	The libraries searched by the OpenVMS Linker to resolve external references. Depending on the needs of your program, you have to specify certain libraries in a specific order so that your program links properly. For more information, see *Chapter 1, "Developing VSI C Programs"*.)
link-time scope	The area in which the OpenVMS Linker recognizes an identifier within a given program. See Also *scope*.
literal	A constant whose value is written explicitly in the program. Literal values have type `int` or `double`, depending on their forms.

	Character constants have type `int`. Floating constants have type `double`. Character-string constants have type array of `char`.)
local variable	A variable declared inside a function body. See Also *internal storage class*.
logical expression	An expression made up of two or more operands separated by a logical operator. Each operand must be a fundamental type or must be a pointer or other address expression. Operands do not have to be the same type. Logical expressions always return 1 or 0 (type `int`) to indicate a true or false value, respectively. Logical expressions are always evaluated from left to right, and the evaluation stops as soon as the result is known.
logical operator	One of the binary operators logical AND (&&) and logical OR (||).
loop	A construct that executes a single statement or a block repeatedly until a given expression evaluates to false. The single statement or block is called the loop body. The C language has three types of loops: one that evaluates the expression before executing the loop body (the `while` statement), one that evaluates the expression after executing the loop body (the `do` statement), and one that executes the loop body a specified number of times (the `for` statement).
lvalue	The address in memory that is the location of an object whose contents can be assigned or modified. In this guide, the term describes a category in C grammar. An expression evaluating to an lvalue is required on the left side of an assignment operator (hence its name) and as the operand of certain other operators, such as the increment (+ +) and decrement (− −) operators. A variable name is an example of an expression evaluating to an lvalue, since its address can be taken (with &), and values can be assigned to it. A constant is an example of an expression that is not an lvalue. See Also *rvalue*.
macro	A text substitution that is defined with the `#define` preprocessor directive and can include a list of parameters. The parameters in the `#define` directive are replaced at compile time with the corresponding arguments from a macro reference encountered in the source text.
main_program option	A tag that can be placed on a separate line between the function parameter list and the rest of a function definition to tell the OpenVMS image activator to begin program execution with this function. You can use the `main_program` identifier when there is no function named `main`; it is not a keyword; it can be spelled in upper- or lowercase; and it is specific to VSI C for OpenVMS systems.
members	Segments of the aggregate data structures (arrays, structures, or unions) that are declared to be of either scalar or aggregate data type. See Also *aggregate*.
module	• The object code produced and placed into a file with a .OBJ extension after a compilation unit has been compiled. The object file is the file name with the .OBJ extension; the object module is

	the system-recognized name (usually the same as the object-file name without an extension). • A segment of object code located in an object library.
multiplication operator	An operator that performs multiplication (*), division (/), or modular arithmetic (%). If necessary, it performs the arithmetic conversions on its operands. The mod operator (%) yields the remainder of the first operand divided by the second.
null pointer	A pointer variable that has not been assigned an lvalue and whose value has been initialized to 0. If you use a null pointer in an expression that needs a value, the compiler will let you try to access memory location 0, which will cause the ACCVIO hardware error. The `NULL` macro can be used when comparing for a null pointer. It is defined in both the `<stdio.h>` and `<stddef.h>` header files as follows: `(void *) 0`.
null character	The escape sequence (\0) that VSI C uses to terminate all character strings. The `NULL` macro can be used when comparing for null characters. It is defined in both the `<stdio.h>` and `<stddef.h>` header files as follows: `(void *) 0`.
object	Data stored at a location in memory represented by an identifier. Objects are one of the basic elements that the language can manipulate; that is, the elements to which operators can be applied. In C, objects include data (such as integers, real numbers, or characters), data structures (arrays, structures, or unions), and functions.)
occlude	In the Curses Screen Management package, when the area of one defined window overlaps the area of another defined window on the terminal screen. See Also *Curses*.
operator	A character that performs an operation on one or more operands. In order of precedence (high to low), operators are classified as the primary-expression operators, unary operators, binary operators, the conditional operator, assignment operators, and the comma operator.
parameter	A variable listed in the parentheses and declared between the function identifier and body in the function definition. The parameter receives a copy of the value of an associated argument when the function is called. The items in parentheses in a macro definition are also called parameters, but the semantics are different from C function calls.)
pointer	A variable that contains the address (lvalue) of another variable or function. A pointer is declared with the unary asterisk operator (*).
portability	The ability to compile an unaltered C source program on several operating systems and machines; in this guide particularly, between UNIX and OpenVMS systems.
pragma	A preprocessor directive that produces implementation-specific results. Certain pragmas may not be portable, but other compilers may support pragmas that are supported by VSI C for OpenVMS systems.

	See Also *preprocessor directives*.
precedence of operators	The order in which operations are performed. If an expression contains several operators, the operations are executed in the following order: primary expression operators, unary operators, binary operators, the conditional operator, assignment operators, and the comma operator.
preprocessor directives	Lines of text in a C source file that change the order or manner of subsequent compilation. The directives are `#define`, for macro substitution and other replacements; `#undef`, to cancel a previous `#define`; `#include`, to include an external source text; `#line`, to specify a line number to the compiler; `#module`, to specify a module name to the linker; `#dictionary`, to extract data structures from the Common Data Dictionary; `#pragma`, to give the compiler implementation-specific information; and `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`, to place conditions on the compilation of sections of a program. In VSI C, these directives are processed by an early phase of the compiler, not by a separate program.)
primary expression	An expression that contains only a primary-expression operator or no operator. Primary expressions include previously declared identifiers, constants, strings, function calls, subscripted expressions, and references to structure or union members.
primary-expression operator	An operator that qualifies a primary expression. The set of such operators consists of paired brackets ([ ]) to enclose a single subscript; paired parentheses (( )) to enclose an argument list or to change the associative precedence of operators; a period (.) to qualify a structure or union name with the name of a member; and an arrow (–>) to qualify a structure or union member with a pointer or other address-valued expression.
program section (psect)	An area of virtual memory that has a name, a size, and a series of attributes that describe the intended or permitted usage of that permanent variable. Variables of type `static`, and of all external and global types are placed in psects. See Also *lifetime*.
refresh	A Curses Screen Management term describing the updating of the terminal screen so that the latest contents of defined windows are placed on the screen. No edits made to any window can appear on the terminal screen until you refresh the window on the screen using `refresh`, `wrefresh`, or `touchwin`. See Also *Curses*.
relational operator	One of the operators less than (<), greater than (>), less than or equal to (<=), or greater than or equal to (>=). The result (which is of type `int`) is 1 or 0, indicating a true or false relation, respectively. If necessary, the arithmetic conversions are performed on the two operands. Relational operators group from left to right.
run-time library	In VSI C for OpenVMS systems, the group of common functions and macros that accompany the compiler that may be called to perform I/O tasks, character-string manipulation, math tasks, system calls, and

	various other tasks. The C language includes no facilities to administer I/O, so compilers include run-time libraries to provide this service. The VSI C Run-Time Library (C RTL) is shipped with the OpenVMS operating system. You can access the C RTL by receiving a copy of the function module in your program's image, or by sharing the function image with your program so that control is passed to the function image and then back to your program. See Also *shareable image*.
rvalue	The object stored at a location in memory represented by an identifier. The rvalue of a variable is the variable's object. See Also *lvalue*, *object*.
scalar	Single objects, including pointers, that can be manipulated in their entirety, in an arithmetic expression. and. See Also *object*, *aggregate*.
scope	The portion of a program in which a particular name has meaning. The link-time scope of names declared in external definitions possibly extends from the point of the definition's occurrence to the end of the program. The scope of the names of function parameters is the function itself. The scope of names declared in any block (that is, after the brace beginning any compound statement) is restricted to that block. Names declared in a block supersede any other declaration of the name, including external definitions, for the extent of that block. Tags within `struct`, `union`, `typedef`, and `enum` declarations are identifiers that are subject to the same scope rules as any identifiers. Member names in structure or union references are not subject to the same scope rules ( ). The scope of a label is the entire function containing the label. See Also *uniqueness*.
shareable image	An OpenVMS image that passes control to another image that passes control back to the original program. You can access the VSI C Run-Time Library (C RTL) as a shared image; control is passed to the C RTL and then back to your program instead of a copy of the function's object module being copied into your program's image.)
shift operator	One of the binary operators (<<) or (>>). Both operands must have integral types. The value of the expression E1<< E2 is the result of expression E1 (interpreted as a bit pattern) left-shifted by E2 bits. The value of E1 >>E2 is E1 right-shifted by E2 bits.
statement	The language elements that perform the action of a function. Statements include expression statements (an expression followed by a semicolon), null statements (the semicolon by itself), compound statements (blocks), and an assortment of statements identified by keywords (such as `return`, `switch`, and `do`).
static storage class	A storage class that permits identifiers to be recognized possibly from the point of the declaration to the end of the compilation unit. Identifiers of the static storage class are declared using the `static` storage-class specifier. See Also *scope*.

stderr	The predefined file pointer associated with the terminal to report run-time errors. The pointed file is equivalent to the OpenVMS logical SYS$ERROR and the file descriptor 2. To use this definition, include the `stdio` definition module in your source code using the `#include` preprocessor directive.)
stdin	The predefined file pointer associated with the terminal to perform input. The pointed file is equivalent to the OpenVMS logical SYS$INPUT and the file descriptor 0. For example, if you specify `stdin` as the pointer to the file to read from in the `getc` macro, the macro reads from the terminal. To use this definition, include the `stdio` definition module in your source code using the `#include` preprocessor directive.
stdout	The predefined file pointer associated with the terminal to perform output. The pointed file is equivalent to the OpenVMS logical SYS$OUTPUT and the file descriptor 1. For example, if you specify `stdout` as the pointer to the file to write to in the `putc` macro, the macro writes to the terminal. To use this definition, include the definition module `stdio` in your source code using the `#include` preprocessor directive.
storage class	The attribute that, with its type, determines the location, lifetime, and scope of an identifier's storage. Examples are `static`, `external`, and `auto`.)
storage-class modifier	Keywords used with the storage-class and data-type keywords to change program section attributes of variables, which restricts access to them. The two storage-class modifiers are `noshare` and `readonly`.
string	• An array of type `char`. • A constant consisting of a series of ASCII characters enclosed in quotation marks. Such a constant is declared implicitly as an array of `char`, initialized with the given characters, and terminated by a null character (ASCII 0, VSI C escape sequence \0).
structure	An aggregate type consisting of a sequence of named members. Each member may have either a scalar or an aggregate type. A structure member may also consist of a specified number of bits called a bit field.
symbolic constant	An identifier assigned a constant value by a `#define` directive. You may use a symbolic constant wherever a literal is valid.
tags	Identifiers that represent a declaration of the data types `struct`, `union`, or `enum`. You may use tags in declarations from that point onward in the program to declare other variables of the same type without having to key in the lengthy declaration again.)
tokens	The fundamental elements making up the text of a C program. Tokens are identifiers, keywords, constants, strings, operators, and other separators. White space (such as spaces, tabs, new lines, and comments) is ignored except where it is necessary to separate tokens.

type	The attribute that, with its storage class, determines the meaning of the values found in the identifier's storage. Types include the integral and floating types, pointers, enumerated types, the `void` data type, and the derived types array, function, structure, and `union`.)
type name	The declaration of an object of a given type that omits the object identifier. A type name is used as the operand of the cast and `sizeof` operators.
unary operator	An operator that takes a single operand. In C, unary operators either precede or follow the operand. The set includes the asterisk (indirection), ampersand (address of), minus (arithmetic unary minus), exclamation (logical negation), tilde (one's complement), double plus (increment), double minus (decrement), cast (force type conversion), and `sizeof` (yields the size, in bytes, of its operand) operators.)
union	A union is an aggregate type that can be considered a structure, all of whose members begin at offset 0 from the base, and whose size is sufficient to contain any of its members. A union can only contain the value of one member at a time.
uniqueness	A property of the names used for certain structure and union members. A name is unique if either of the following conditions is true: • The name is used only once. • The name is used in two or more different structures (or unions), but each use denotes a member at the same offset from the base and of the same data type. The significance of uniqueness is that a unique member name can possibly be used to refer to a structure in which the member name was not declared (although a warning message is issued).
variable	An identifier used as the name of an object.
value	The result of an expression. For example, when a variable on the right side of an assignment expression is evaluated, the value obtained is the object (rvalue) of the variable; when a variable on the left side of an assignment expression is evaluated, the value obtained is the address (lvalue) of the variable.
white space	Spaces, tabs, new lines, and comments. The compiler defines where you can and cannot place these characters.)
windows	In the Curses Screen Management package, the defined rectangular regions on the terminal screen that you can write upon, rearrange, move to new positions on the screen, and delete from the screen. You define windows by specifying the upper left corner coordinate, the number of lines, and the number of columns comprising the window. To see the results after editing a window, you must refresh the window on the terminal screen. See Also *refresh*.